FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Performance-Analyse paralleler Löser für dünnbesetzte lineare Gleichungssysteme auf IBM p690

Sandra Vogt

FZJ-ZAM-IB-2005-03

März 2005

(letzte Änderung: 8.3.2005)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Performance-Analyse paralleler Löser für dünnbesetzte lineare Gleichungssysteme auf IBM p690

Das Lösen großer dünnbesetzter linearer Gleichungssysteme ist ein häufig auftretendes Problem in wissenschaftlichen und technischen Anwendungen. Hierfür existieren zwei wesentlich verschiedene Klassen von Methoden - die iterativen und die direkten Methoden. Bei den direkten Verfahren zum Lösen eines linearen Gleichungssystems wird eine explizite Faktorisierung der Systemmatrix durchgeführt. Dies ergibt besondere Vorteile, wenn dasselbe Gleichungssystem mit mehreren rechten Seiten gelöst werden muss, da die Faktorisierung nur einmal erforderlich ist. Bei den iterativen Lösern lässt sich hingegen die Dünnbesetztheit der Systemmatrix meist besser ausnutzen.
Anhand des Problems wird entschieden, welche Methode am besten geeignet ist. Bei linearen Gleichungssystemen aus dem technischen Anwendungsbereich, die mit der Methode der Finiten Elemente erzeugt wurden, sind die Systemmatrizen meist symmetrisch positiv definit und dünn besetzt, daher ist für ihre Lösung das direkte Verfahren der Cholesky-Zerlegung gut geeignet.
Die Lösung in der Praxis auftretender dünnbesetzter Gleichungssysteme erfordert zudem häufig sehr viel Zeit und Speicherplatz. Daher wird sie bevorzugt auf Parallelrechnern durchgeführt. Hierfür stehen umfangreiche Softwarepakete zur Verfügung. Eines davon ist das „Watson Sparse Matrix Package“, welches auf dem IBM p690 Cluster „Jump“ (JUelich Multi Processor) bereitgestellt wird. Es bietet verschiedene Routinen zur Cholesky-Faktorisierung dünnbesetzter symmetrisch positiv definiter Matrizen. Die Parallelisierung erfolgt wahlweise unter Verwendung des gemeinsamen Speichers auf der Grundlage von Threads oder für verteilten Speicher mit Nachrichtenaustausch.
Im Rahmen der vorliegenden Arbeit wurde die Performance dieses Softwarepakets für die Lösung dünnbesetzter linearer Gleichungssysteme mit symmetrisch positiv definiten Matrizen untersucht. Dabei wurden die angebotenen unterschiedliche Parallelisierungsmethoden und Verfahren zur Faktorisierung der Systemmatrizen verglichen.

Performance-Analysis of Parallel Solvers for Sparse Linear Systems on IBM p690

In various areas of scientific computing, it is often required to solve large sparse systems of linear equations. The existing algorithms decompose to direct and iterative methods. When a direct method is used, the system matrix is explicitely factorized. This is advantageous, if the linear system has to be solved for multiple right hand sides, since the factorization has to be done only once. However, iterative solvers can better exploit the sparsity of a matrix.
Depending on the problem, the most suitable method is chosen. The coefficient matrices of linear systems rising from computations using the finite element method mostly are symmetric positive definite and sparse. Here the Cholesky factorization, a direct method, is a good choice.
For the solution of very large sparse linear systems, often parallel computing can be mandatory in terms of computing time and storage requirements. The Watson Sparse Matrix package, installed on the IBM p690 cluster Jump (JUelich Multi Processor), offers several parallel routines for the Cholesky factorization of sparse symmetric positive definite matrices. Alternatively, parallelization based on threads for shared memory or based on message passing for distributed memory can be used.
In the frame of this report, the performance of the above mentioned software package was investigated for the solution of sparse linear systems of equations with symmetric positive definite matrices. The different parallelization strategies and methods for the factorization of the system matrix were compared.

Kapitel 1

Einleitung

Das Lösen großer dünnbesetzter linearer Gleichungssysteme ist ein häufig auftretendes Problem in wissenschaftlichen und technischen Anwendungen. Hierfür wurden zahlreiche Verfahren entwickelt. Dabei werden zwei Arten von Methoden unterschieden - die iterativen und die direkten Methoden. Anhand des Problems wird entschieden, welche Methode am besten geeignet ist.
Bei direkten Methoden zum Lösen eines dünnbesetzten linearen Gleichungssystems wird eine explizite Faktorisierung der Matrix durchgeführt. Sie sind für die Lösung mit mehreren rechten Seiten eine besonders effektive Lösungsvariante, da die Faktorisierung unabhängig von der Zahl der rechten Seiten nur einmal durchgeführt werden muss. Dies verringert die Rechenzeit bei mehreren rechten Seiten wesentlich. Die direkten Methoden sind außerdem sehr robust und allgemein gehalten. Bei linearen Gleichungssystemen mit dünnbesetzten Matrizen vor allem aus dem Bereich der Strukturmechanik sind sie meist die sinnvollste Lösungsmethode. In diesem Fall bilden die Koeffizienten der linearen Gleichungssysteme meist symmetrisch positiv definite Matrizen, für deren Lösung die Cholesky-Zerlegung gut geeignet ist.
Im Gegensatz zu den direkten Verfahren kann bei iterativen Verfahren die Dünnbesetztheit der Matrix besser ausgenutzt werden. Eine Untersuchung der iterativen Verfahren soll jedoch nicht Gegenstand dieser Arbeit sein. Sie soll sich auf die Cholesky-Zerlegung beschränken.
Da in der Praxis auftretende dünnbesetzte Gleichungssysteme häufig zum Lösen einen großen Zeit- und Speicherplatzbedarf benötigen, werden diese bevorzugt auf Parallelrechnern ausgeführt.

Die Aufgabe der hier vorliegenden Arbeit besteht darin, die Performance für die Lösung praxisrelevanter, großer dünnbesetzter linearer Gleichungssysteme mit symmetrisch positiv definiten Matrizen bei Verwendung direkter Lösungsverfahren zu untersuchen. Wegen des bereits erwähnten großen Zeit- und Speicherplatzbedarfs können die Gleichungssysteme nicht auf einem herkömmlichen PC gelöst werden. Daher wurden die Berechnungen auf dem neuen IBM Regatta System des Forschungszentrums Jülich, einem parallelen Höchstleistungsrechner, durchgeführt. Seine Architektur wird in Kapitel 2 beschrieben.
Es sollen Einblicke in die Performance des Lösens der Gleichungssysteme in Abhängigkeit von unterschiedlichen Faktoren gewonnen werden. Dabei spielt insbesondere die Ausführungszeit für die Lösung des Problems eine entscheidende Rolle, die durch Einsatz von Parallelisierungstechniken verringert werden kann. Unterstützt wird die Auswertung durch das HPM-Toolkit, welches außer der Ausführungszeit auch Informationen über die Anzahl der Rechenoperationen liefert. Die Ergebnisse werden mit Hilfe von Gsharp und Maple graphisch ausgewertet. Die verwendeten Tools werden in Kapitel 3 erläutert.
Auf dem IBM Regatta System steht das Programmpaket „Watson Sparse Matrix Package“ (WSMP) zur Verfügung. Es bietet verschiedene Lösungsroutinen für große lineare Gleichungssysteme mit dünnbesetzten Matrizen, wobei direkte Verfahren verwendet werden. Diese Bibliothek nutzt die Dünnbesetztheit und die Symmetrie von Matrizen aus, um Speicherplatz zu sparen, was bei sehr

großen Systemen von enormer Bedeutung sein kann. Das Vorgehen und die verschiedenen Lösungsansätze sowie die Parallelisierungsstrategie der Bibliothek werden in Kapitel 4 vorgestellt.
Um die Performance-Untersuchungen durchführen zu können, werden verschiedene Datensätze zum Zeitmessen benötigt. Zum Einen stehen Datensätze aus der Praxis, wie die Harwell-Boeing-Matrizen zur Verfügung, zum Anderen selbst erstellte Datensätze, die auf Hexe27-Elementen basieren, da die Harwell-Boeing-Matrizen nur mit begrenzter Dimension zur Verfügung stehen. Beide Arten von Datensätzen werden in Anhang A erläutert.
Um abschließend zu einem Ergebnis der Performance-Untersuchungen zu kommen, werden in Kapitel 5 die Zeitmessungen zu den verschiedenen Verfahren und den unterschiedlichen Parallelisierungsmethoden - Threads-basiert auf gemeinsamem Speicher oder MPI-basiert auf verteiltem Speicher - graphisch dargestellt. Untersucht wird auch die Auslastung der einzelnen Prozessoren bei den parallelen Rechnungen sowie die Skalierbarkeit der Verfahren.

Kapitel 2

Rechnerarchitekturen

2.1 Motivation

Zum Lösen großer linearer Gleichungssysteme, wie sie in dieser Arbeit vorliegen, wird viel Rechenzeit benötigt. Um die Verfahren zum Lösen zu verkürzen, werden die Berechnungen parallelisiert. Dafür sind parallele Rechnerarchitekturen nötig, da dies auf einem einfachen Einprozessorsystem, wie beispielsweise einem herkömmlichen PC, nicht möglich ist.

2.2 Aufbau eines Parallelrechners

Die drei wesentlichen Komponenten eines Parallelrechners sind die Prozessoren, der Hauptspeicher und das zur Verbindung benötigte Kommunikationsnetzwerk. Aus diesen Komponenten lassen sich unterschiedliche Parallelrechner zusammenstellen. Dabei unterscheidet man zwischen Shared Memory und Distributed Memory Systemen.

2.2.1 Distributed Memory Systeme

Beim Distributed Memory System steht jedem Prozessor sein eigener Speicher zur Verfügung (Abbildung 2.1). Um Daten zwischen den Prozessoren austauschen zu können, müssen die verschiedenen Prozessoren miteinander kommunizieren (= explizite Kommunikation (Message Passing)). Sie senden sich gegenseitig Nachrichten, um beispielsweise mitzuteilen, wann mit einem bestimmten Schritt begonnen wird oder einer endet. Desweiteren senden sich die Prozessoren Daten zu, die ein anderer Prozessor für seine Berechnungen benötigt. Häufig übernimmt ein Prozessor, der Master, die “Leitung“. Er verteilt die Daten auf die verschiedenen Prozessoren und sammelt diese am Ende auch wieder ein. Er sorgt durch Nachrichtenverteilung für einen geregelten Ablauf des Programms.

Programmiertechniken

Auf Distributed Memory Maschinen wird meist das sogenannte SPMD (**S**ingle **P**rogram **M**ultiple **D**ata) [1] Programmiermodell verwendet. Hierbei wird auf jedem Prozessor dasselbe Programm gestartet, es rechnet jedoch mit den jeweils lokalen Daten. Durch Verzweigungen im Programmcode, in Abhängigkeit von der Prozessornummer, kann MPMD (**M**ultiple **P**rogram **M**ultiple **D**ata) erreicht werden. Dabei müssen sich die einzelnen Prozesse gegenseitig identifizieren um miteinander kommunizieren zu können.

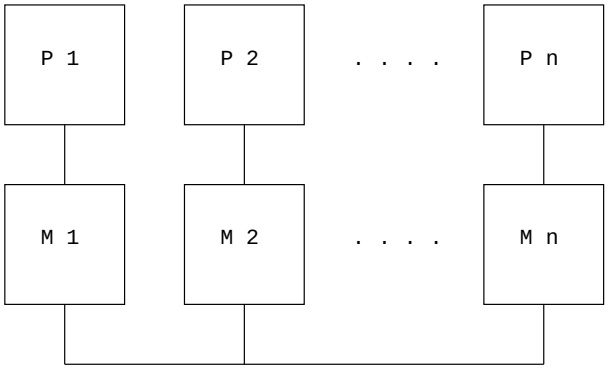

Abbildung 2.1: Graphische Darstellung des Distributed Memory Systems

Für die explizite Kommunikation, die beim Distributed Memory System benötigt wird, wird eine Kommunikationsbibliothek mit Routinen zum Versenden und Empfangen von Daten und zur Synchronisation von Prozessen gebraucht. Die zur Zeit am häufigsten verwendete Bibliothek ist MPI (**M**essage **P**assing **I**nterface) [2]. Ein mögliches Problem der MPI-Programmierung ist beispielsweise der "Deadlock". Dieser entsteht zum Beispiel wenn alle Prozessoren auf eine Nachricht warten, und dadurch keiner weiter arbeiten kann, oder wenn ein Prozessor auf Grund eines falschen Ergebnisses einen globalen Kommunikationsoperator nicht erreicht. Der Vorteil von MPI ist die Portabilität und die hohe Übertragungsleistung durch hoch optimierte Implementierungen. Die Nutzung von MPI ist auch bei gemeinsam verfügbarem Speicher, also auf einem Shared Memory System, möglich, und es gibt viele MPI-basierte Bibliotheken, wie zum Beispiel die WSMP-Bibliothek, die im folgenden Kapitel beschrieben wird.

2.2.2 Shared Memory Systeme

Bei einem Shared Memory System teilen sich alle Prozessoren einen gemeinsamen Hauptspeicher (Abbildung 2.2). Der Zugriff auf den Hauptspeicher erfolgt durch ein Kommunikationsnetzwerk, welches auch dafür sorgen muss, dass bei der Verwendung von lokalen Caches die Konsistenz der Daten sichergestellt wird. Der Hauptspeicher, der gegebenenfalls aus mehreren Komponenten besteht, besitzt einen einzigen Adressraum, auf den jeder Prozessor zugreifen kann. Dabei muss darauf geachtet werden, dass Daten, die später noch von anderen Prozessen gebraucht werden, nicht geändert werden.

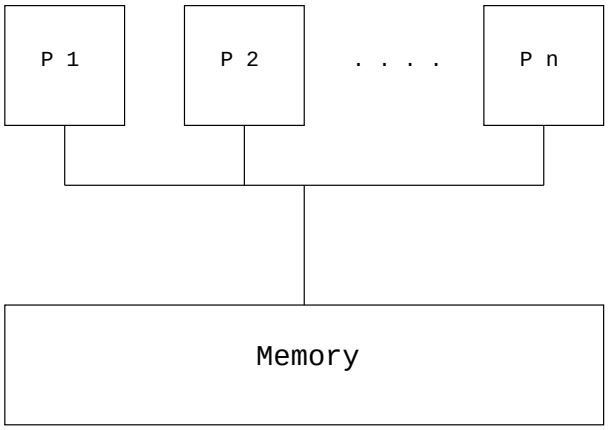

Abbildung 2.2: Graphische Darstellung des Shared Memory Systems

Programmiertechniken

Auch bei Shared Memory Maschinen wird das SPMD Programmiermodell benutzt. Im Gegensatz zum Distributed Memory verzweigt sich hier das Programm in mehrere sogenannte „Threads“, die parallel auf unterschiedlichen Daten dieselben Operationen ausführen. Die verschiedenen Daten sind dabei unterschiedlichen Bereichen des gemeinsamen Hauptspeichers zugeteilt.
Beim Programmieren kann aufgrund des gemeinsamen Hauptspeichers wie bei einem Programm auf einem Einprozessorsystem vorgegangen werden. Eine Möglichkeit dieser Vorgehensweise ist OpenMP-Programmierung [3]. OpenMP ist ein Standard, der einen Satz von Pragmas an den Compiler liefert. Das sequentielle Programm wird mit Compiler-Direktiven versehen, die die parallelen Teile und die privaten und globalen Variablen markieren. Diese Compiler-Direktiven werden von einem Compiler auf einem Einprozessorsystem überlesen, da sie für ihn einen Kommentar darstellen. Somit kann solch ein Programm sowohl auf einem Ein- als auch auf einem Mehrprozessorsystem verwendet werden. Der Compiler auf einem Mehrprozessorsystem sorgt dann bei der Parallelisierung für einen geregelten Zugriff auf die im Hauptspeicher abgelegten Daten.
Bei dieser Art der Parallelisierung werden *DO*-Schleifen oder implizite Schleifen parallelisiert. Dabei muss aber darauf geachtet werden, dass in einer Iteration keine Feldkomponenten geändert werden, die in einer späteren Iteration noch benötigt werden. Desweiteren dürfen keine skalaren Variablen geändert werden, die in einer weiteren Iteration oder nach der Schleife noch gebraucht werden. Dabei sollte der Overhead der Parallelisierung im Vergleich zur Rechenarbeit in der Schleife nicht zu hoch sein [4].

2.3 Konfiguration des IBM Supercomputers

Der IBM Supercomputer Jump (**Ju**elich **m**ulti**p**rocessing) des Forschungszentrums Jülich ist eine Kombination aus Shared Memory und Distributed Memory System. Somit kann sowohl mit Threads oder OpenMP, welches für solche Shared Memory Systeme ausgelegt ist, als auch mit MPI, welches wie oben beschrieben, auch auf Shared Memory Systemen benutzt werden kann, gearbeitet werden.
Bei dem IBM Supercomputer Jump handelt es sich um eine Clusterarchitektur, die aus 41 IBM p690 Knoten - sogenannten „Compute-Nodes“ - besteht. Dabei ist zu beachten, dass nur innerhalb eines Knotens Shared Memory Programmierung möglich ist. Zwischen den Knoten muss ein Programmiermodell für verteilten Speicher (Distributed Memory), zum Beispiel MPI, verwendet werden. Jeder Compute-Node besteht aus 32 Power4+ Prozessoren mit einer Taktrate von 1,7 GHz und einem gemeinsamen Hauptspeicher von 128 GB. Jeder dieser Knoten verfügt pro Prozessor über einen internen Level1-Cache mit 32 KB Data- und 64 KB Instruktionsspeicher, pro Chip (= 2 Prozessoren) über einen Shared Level2-Cache mit 1,5 MB und pro Knoten über einen Shared Level3-Cache mit 512 MB. Jeder Prozessor besitzt zwei Multiply-Add Rechenwerke, so dass pro Takt bis zu 4 Floating Point Operationen ausgeführt werden können ($4 * 1,7 * 32 = 218$ GFLOPS je Knoten).
Das Gesamtsystem besteht aus 1312 Prozessoren und erreicht somit eine Peak-Performance von 8,9 TFLOPS bei einem Hauptspeicher von 5,2 TB. Die einzelnen Knoten des Supercomputers sind durch einen High-Performance Switch verbunden.
Zum Zeitpunkt der Auswertung stand dem Benutzer das Betriebssystem AIX 5.2 zur Verfügung [5].

Kapitel 3

Verwendete Software

3.1 Motivation

Für die Entwicklung der dieser Arbeit zugrundeliegenden Programme und für die spätere Performance-Untersuchung wird unterschiedliche Software benötigt. Insbesondere zu erwähnen ist die Bibliothek WSMP (**W**atson **S**parse **M**atrix **P**ackage), die in diesem Kapitel kurz beschrieben wird und auf die im folgenden Kapitel näher eingegangen wird.
Damit Performance-Untersuchungen durchgeführt werden können, sind Zeitmessungen nötig. Dafür stehen Zeitmessroutinen und unterschiedliche Performance-Tools zur Verfügung, deren Output zur besseren Veranschaulichung graphisch aufbereitet wird.

3.2 Gleichungssystemlöser

Zum Lösen der dünnbesetzten, symmetrisch positiv definiten Gleichungssysteme wurden Routinen aus der Bibliothek WSMP (**W**atson **S**parse **M**atrix **P**ackage) Version 1.9.4 verwendet, welche ein spezielle IBM-Implementation einer ähnlichen public-domain Software ist. Für diese spezielle IBM-Version steht kein Quellcode zur Verfügung.
Beim Speichern der Matrizen wird die Dünnbesetztheit ausgenutzt, um Speicherplatz und Rechenoperationen zu sparen. Die genaue Funktionsweise der verschiedenen Löser wird in Kapitel 4 erklärt [6].

3.3 Programmiertools

3.3.1 Compiler

Das in dieser Arbeit verwendete Programm zur Analyse der Performance der Routinen aus der Bibliothek WSMP wurde in Fortran entwickelt. Auf dem IBM Server p690 wurde der Fortran-Compiler mpxlf90_r Version 8.1.1 verwendet.

3.3.2 LoadLeveler

LoadLeveler (Version 3.2.0.7) ist ein Job-Management-System, welches dem Benutzer erlaubt, mehrere Jobs simultan zur interaktiven oder zur Batch-Verarbeitung abzuschicken. Der LoadLeveler teilt die Jobs ein und stellt Hilfsmittel zum Erstellen und Abschicken dieser Jobs bereit [7].
Die Einteilung der Jobs erfolgt nach bestimmten Kriterien. Ausschlaggebend dabei sind beispielsweise die Priorität eines einzelnen Jobs, d.h. ob dieser gegenüber einem anderen Job bevorzugt behandelt wird, oder auch der Speicherbedarf des Jobs, die Anzahl der Prozessoren die genutzt

werden bzw. das gewählte Zeitlimit für die Ausführung des Jobs.
Das Abschicken eines Jobs kann auf zwei Arten erfolgen:

- Bei der interaktiven Variante wird das kompilierte Programm mit *llrun* gestartet. Zum Entwickeln und Testen wurde bei dieser Arbeit *llrun* in der Betaversion 1.3 verwendet. Bei der Verwendung von *llrun* kann die Laufzeit beeinträchtigt werden, da auf einem Knoten zusätzlich Jobs anderer Benutzer abgearbeitet werden. Dies war beim Entwickeln noch nicht von Interesse, sondern erst bei der Durchführung der Zeitmessungen. Bei dem Starten eines Programms mit *llrun* kann, je nach Programmiermodell, die Anzahl der Prozessoren bzw. der Threads, sowie die Anzahl der Knoten, auf denen gerechnet werden soll, angegeben werden. Standardmäßig wird auf einem Knoten gearbeitet. Beispielsweise wird mit dem folgenden Aufruf

  llrun -n 1 -t 1 -p 16 a.out

  auf einem Knoten mit 16 Prozessoren MPI-basiert gerechnet. Dabei gibt *-n* die Anzahl der Knoten, *-t* die Anzahl der Threads und *-p* die Anzahl der Prozessoren an.

- Für die Zeitmessungen wird Batchverarbeitung verwendet. Dabei kann mit Hilfe von *llsubmit* ein Job an den *LoadLeveler* übergeben werden. Der Vorteil dabei ist, dass ein Knoten für die Berechnung reserviert werden kann, wodurch Verfälschungen bei der Zeitmessung vermieden werden. Hierfür wird ein Batchfile erstellt, das Variablen enthält, die die Eigenschaften des Batchjobs beschreiben. Wichtig für diese Arbeit sind
  - *node*: Anzahl der Knoten, auf denen gerechnet werden soll
  - *tasks_per_node*: Anzahl der benutzten Prozesse
  - *ConsumableCpus*: Anzahl der benutzten Threads
  - *node_usage*
    = shared: Jobs anderer Benutzer können gleichzeitig auf dem Knoten bearbeitet werden oder
    = not_shared: ausschließliche Verwendung des Knotens durch den Benutzer
  - *ConsumableMemory*: der zur Verfügung stehende Speicher
  - *wall_clock_limit*: maximale Ausführungszeit
  - *data_limit*: Speicherplatz, der zum Speichern der Daten zur Verfügung steht
  - *stack_limit*: Speicherplatz für den Stack
  - *output*: Name der Ausgabedatei
  - *error*: Name der Datei, in die auftretende Fehler geschrieben werden
  - *OMP_NUM_THREADS*: Hier muss nochmals die Anzahl der benutzten Threads angegeben werden

  Am Ende des Batchfiles erfolgt mit **poe a.out** der Aufruf des ausführbaren Programms.

3.4 Zeitmessung und Performance-Untersuchung

Um Performance-Untersuchungen durchführen zu können, werden Zeitmessroutinen und Performance-Tools benötigt. So können unter verschiedenen Gesichtspunkten wie Problemgröße, Programmiermodell und Prozessoranzahl, Erkenntnisse gewonnen werden.

3.4.1 Zeitmessroutinen

Die Zeitmessroutine **rtc()** (**R**eal **T**ime **C**lock) wird in dem Programmcode dieser Arbeit verwendet. Sie gibt die Zeiten an, die für die unterschiedlichen Abschnitte zur Lösung des linearen Gleichungssystem benötigt werden. Desweiteren können die verschiedenen Algorithmen zur Lösung des Glei-

chungssystems damit verglichen werden.
Die Zeitmessroutine **rtc()** gibt die Realzeit in Sekunden an. Die Auflösung beträgt $0.1\mu s$.

3.4.2 Hardware Performance Monitor (HPM) Toolkit

Das **HPM** Toolkit [8] wurde für Performance-Messungen auf IBM-Rechnern entwickelt. Es besteht aus:

- dem **hpmcount**, der eine Anwendung startet und nach der Ausführung die benötigte Zeit, Hardware Metriken (beispielsweise Flip) sowie abgeleitete Größen (wie MFlip/sec) und Statistiken zur Auslastung liefert. Flip ist dabei ein spezielles Maß von IBM. Bei Power4 Prozessoren, wie sie in Jump vorliegen, gilt:
  „Flip = FPU 0 instructions + FPU 1 instructions + FMAs executed - FPU Stores“ [8],
  also die Anzahl der Instruktionen beider Rechenwerke plus die Anzahl der Multiply-Add Instruktionen, da eine solche Instruktion für zwei Rechenoperationen steht. Die FPU-Speicherungen werden bei der Power4-Architektur zunächst als Instruktionen gezählt und werden dann wieder abgezogen, um mit Flip letztendlich die Anzahl der Floating Point Instruktionen, üblicherweise Flop genannt, zu erhalten.
- **libhpm**, einer Instrumentierungsbibliothek, mit deren Hilfe in einem Programm einzelne Regionen instrumentiert werden können, und die dann für jede instrumentierte Region eine zusammengefasste Ausgabe der oben genannten Angaben liefert. Der Vorteil im Vergleich zu *hpmcount* ist, dass man die Angaben für unterschiedliche Programmteile und nicht nur für das gesamte Programm erhält. Allerdings ist die Nutzung aufwendiger, da das Programm instrumentiert und einige Änderungen im Makefile vorgenommen werden müssen.
- dem graphischen Interface **hpmviz**. Diese graphische Benutzerschnittstelle visualisiert die von *libhpm* generierte Performance-Datei.

In der vorliegenden Untersuchung wurde *libhpm* benutzt, da die unterschiedlichen Laufzeiten verschiedener Programmteile untersucht werden sollten und dies, wie oben beschrieben, mit *hpmcount* nicht möglich ist.

3.5 Graphische Darstellung der Ergebnisse

Um die aus der Zeitmessung und dem Performance-Tool *HPM* gewonnenen Ergebnisse auswerten zu können, stellt man diese graphisch dar. Dafür werden zunächst aus den Messergebnissen Tabellen erstellt. Diese können anschließend mit einem Graphikprogramm eingelesen, bearbeitet und zuletzt graphisch dargestellt werden.

3.5.1 Gsharp

Gsharp [9] ist eine Software zur Erstellung von 2D- und 3D-Präsentationsgraphiken, die sowohl interaktiv mit einer Point-and-Click-Motivoberfläche, als auch im Batch über eine Kommandoschnittstelle erstellt werden können.
Gsharp verwendet die Script-Sprache *GSL*. Dabei handelt es sich um eine lesbare ASCII-Datei, die interaktiv oder durch Programmierung erstellt wird.
In dieser Arbeit wurde mit Point-and-Click ein graphisches Objekt generiert. Mit Hilfe des Script Builders wurden die dem Objekt entsprechenden *GSL* Befehle als wiederverwendbares Script gespeichert. Dieses wurde dann den Eingabedaten entsprechend für die unterschiedlichen Graphiken modifiziert.

3.5.2 Maple

Maple [10] ist ein mathematisches Softwaresystem, welches symbolisches Rechnen (Computeralgebra), numerisches Rechnen und Visualisierung in einer integrierten und benutzerfreundlichen Arbeitsumgebung ermöglicht.
Mit Hilfe von Maple können Funktionen berechnet und später mit Hilfe der plot-Funktion graphisch dargestellt werden. In dieser Arbeit wird die Version 9.5 verwendet.

Kapitel 4

Beschreibung des Algorithmus

4.1 Einleitung

Die numerische Lösung von großen dünnbesetzten linearen Gleichungssystemen, die häufig Mittelpunkt groß angelegter wissenschaftlicher und technischer Berechnungen ist, ist auch Grundlage dieser Arbeit. Die Entwicklung der *Multifrontal Methode* im Jahre 1983 durch Duff und Reid [11] war entscheidend für die Anwendung direkter Methoden bei der Lösung von Gleichungssystemen mit dünnbesetzten Matrizen. Die Multifrontal Methode zerlegt das allgemeine Problem der Faktorisierung einer dünnbesetzten Matrix in eine Folge von partiellen Faktorisierungen von kleinen, dichter besetzten Matrizen (Frontal-Matrizen).
Die Methode hat sich als sehr wertvoll erwiesen. Auch hat sie in den letzten Jahren an Bedeutung gewonnen, wie sich an der großen Akzeptanz und dem häufigen Auftreten in vielen wissenschaftlichen und technischen Applikationen zeigt. Die Methode wird zum Beispiel in vielen groß angelegten Finite Element Berechnungen benutzt.
Der Begriff „multifrontal“ wurde von Duff und Reid benutzt, nachdem sie die Methode als eine Verallgemeinerung der Frontal Methode von Irons entwickelt hatten.
Die in dieser Arbeit benutzte Bibliothek WSMP (**W**atson **S**parce **M**atrice **P**ackage) basiert auf dieser Methode. In der Bibliothek sind sowohl Routinen für Shared Memory Parallelisierung mit Threads, als auch für MPI-Parallelisierung implementiert. Für die Lösung der Gleichungssysteme stehen unterschiedliche Routinen zur Verfügung: Cholesky-Zerlegung, LDL^T-Faktorisierung und ein sogenannter Expert Driver, welcher LDL^T-Faktorisierung, Lösen der Dreieckssysteme und Nachiteration kombiniert. Näheres hierzu findet sich in Kapitel 5.

Im Weiteren wird die Vorgehensweise der Cholesky-Zerlegung beschrieben. Auf die Beschreibung der LDL^T-Faktorisierung wird verzichtet, da sie wie Cholesky funktioniert, nur dass das Wurzelziehen umgangen wird. Daher eignet es sich auch für positiv-semidefinite Matrizen.
Sinn der Frontal Methode ist es, die Rechenschritte des Cholesky-Verfahrens geschickt anzuordnen. Die Vorgehensweise wird in diesem Kapitel beschrieben. Dabei wird die Lösung des linearen Gleichungssystems in drei Phasen unterteilt: Analysieren, Faktorisieren, Lösen. Die Vorgehensweise der verschiedenen Phasen wird in Kapitel 4.2 vorgestellt.
Als Grundlage dienten [12], [13], [6], [14] und [15].

4.1.1 Cholesky-Zerlegung

Die Systemmatrizen der Gleichungssysteme, die in dieser Arbeit gelöst werden, sind symmetrisch positiv definit. Daher wird die Cholesky-Faktorisierung von $n \times n$-Matrizen $A = LL^T$ betrachtet. Ist A eine dünnbesetzte, symmetrisch positiv definite Matrix, so enthält der Cholesky-Faktor L in der Regel mehr Nicht-Null-Elemente als A selbst. Der Grad der Auffüllung hängt von der Reihenfolge ab, in der die Diagonalelemente als Pivotelemente gewählt werden. Dabei kann die

Faktorisierung auf viele unterschiedliche Arten ausgeführt werden, abhängig von der Reihenfolge in der auf die Matrixeinträge zugegriffen wird oder diese aktualisiert werden.
Im folgenden Abschnitt wird nun für eine vollbesetzte Matrix das *Outer-Product*-Verfahren beschrieben.
Ein Schritt des Outer-Product-Verfahrens der Cholesky-Faktorisierung sieht folgendermaßen aus:

$$A = \begin{pmatrix} d & v^T \\ v & C \end{pmatrix} = \begin{pmatrix} \sqrt{d} & 0 \\ v/\sqrt{d} & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & C - vv^T/d \end{pmatrix} \begin{pmatrix} \sqrt{d} & v^T/\sqrt{d} \\ 0 & I \end{pmatrix},$$

wobei d der erste Diagonaleintrag und v ein (n-1)-Vektor ist. Die Submatrix $C - vv^t/d$ ist der Anteil, der zum Faktorisieren übrig bleibt, und dessen Faktorisierung rekursiv, durch Nutzen der gleichen Technik, ausgeführt werden kann. Die erste Spalte der Matrix

$$\begin{pmatrix} \sqrt{d} & 0 \\ v/\sqrt{d} & I \end{pmatrix}$$

bildet die erste Spalte der Matrix L der Cholesky-Zerlegung.
Dieser eine Schritt der Faktorisierung kann einfach auf die Blockform verallgemeinert werden. Dann ist

$$A = \begin{pmatrix} B & V^T \\ V & C \end{pmatrix} = \begin{pmatrix} L^B & 0 \\ VL^{-T} & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & C - VB^{-1}V^T \end{pmatrix} \begin{pmatrix} L_B^T & L_B^{-1}V^T \\ 0 & I \end{pmatrix}, \qquad (4.1)$$

wobei $B = L_B L_B^T$ die Cholesky-Faktorisierung der vorderen $(j-1) \times (j-1)$-Hauptsubmatrix B ist. Die Matrix $C - VB^{-1}V^T$ aus der Gleichung (4.1) muss schließlich in einem weiteren Schritt faktorisiert werden.
Man beachte, dass die $(n-j+1) \times (n-j+1)$-Submatrix ($-VB^{-1}V^T$) die Updates der ersten $j-1$ Zeilen und Spalten der Submatrix C bilden. Diese Update-Matrix kann ausgedrückt werden über die ersten $j-1$ Spalten der Cholesky-Zerlegung:

$$-VB^{-1}V^T = -(VL_B^{-T})(L_B^{-1}V^T) = -\sum_{k=1}^{j-1} \begin{pmatrix} l_{j,k} \\ \vdots \\ l_{n,k} \end{pmatrix} \begin{pmatrix} l_{j,k} & \dots & l_{n,k} \end{pmatrix}.$$

Mit der Multifrontal Methode wird ein Algorithmus definiert, um diese Updates im Fall dünnbesetzter Matrizen zu handhaben.

4.1.2 Der extend-add-Operator

Im Folgenden wird der sogenannte *extend-add*-Operator verwendet. Dieser soll nun anhand eines Beispiels erklärt werden.
Sei R eine $r \times r$-Matrix mit $r \leq n$ und S eine $s \times s$-Matrix mit $s \leq n$. Jede Zeile und Spalte von R und S entspricht einer Zeile bzw. einer Spalte der $n \times n$-Matrix A. Seien $i_1 \leq ... \leq i_r$ die Indexmenge von R aus A und $j_1 \leq ... \leq j_r$ die Indexmenge von S.
Man betrachte nun die Vereinigung der beiden Index-Mengen. Sei $k_1 \leq ... \leq k_m$ die resultierende Menge. Die Matrix R kann erweitert werden, um der Indexmenge $\{k_1, ..., k_m\}$ zu entsprechen, indem eine Anzahl an Null-Zeilen und Null-Spalten eingefügt wird. Auf die gleiche Weise kann S angeglichen werden. $R \dotplus S$ wird definiert durch die $m \times m$-Matrix M, die daraus entsteht, dass die erweiterten Matrizen R und S addiert werden. Man bezeichnet diesen Matrixoperator „$\dotplus$“ als Matrix-*extend-add*-Operator. Diese verallgemeinerte Matrixoperation wird auch als Matrix-Superposition

bezeichnet.
Sei beispielsweise

$$R = \begin{pmatrix} p & q \\ u & v \end{pmatrix}, \quad S = \begin{pmatrix} w & x \\ y & z \end{pmatrix},$$

wobei $\{5,8\}$ und $\{5,9\}$ die Indexmengen von R und S sind. Dann ist $R \dot{+} S$ die folgende 3×3-Matrix, mit der Indexmenge $\{5,8,9\}$:

$$R \dot{+} S = \begin{pmatrix} p & q & 0 \\ u & v & 0 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} w & 0 & x \\ 0 & 0 & 0 \\ y & 0 & z \end{pmatrix} = \begin{pmatrix} p+w & q & x \\ u & v & 0 \\ y & 0 & z \end{pmatrix}.$$

4.2 Beschreibung der Hauptfunktionen

Das Lösen des linearen Gleichungssystems

$$Ax = b,$$

wobei A in unserem Fall eine symmetrisch positiv definite Matrix ist, wird in drei Teile aufgeteilt. Dabei sind die Hauptfunktionen (siehe Abbildung 4.1) das Analysieren, d.h. es wird eine symbolische Faktorisierung durchgeführt, die eigentliche Faktorisierung von A (zum Beispiel mit dem Cholesky-Verfahren) und die Lösung der daraus entstehenden Dreieckssysteme durch Rückwärts- und Vorwärts-Substitution.

Analysieren
Faktorisieren
Loesen

Abbildung 4.1: Die Hauptfunktionen zum Lösen des Gleichungssystems

Diese drei Schritte werden im folgenden Kapitel beschrieben.

4.2.1 Analyse

Bei der symbolischen Faktorisierung wird aus der Struktur der Matrix, nicht aus den Werten, eine Permutationsmatrix P erzeugt, die für möglichst wenig Fill-in in L sorgt.
Daneben wird in der Analyse-Phase der Eliminations-Baum erstellt, der im wesentlichen zur Steuerung der Faktorisierungs- bzw. Lösungsphase benutzt wird. Der Eliminations-Baum der Matrix A ist definiert als Struktur mit n Knoten $\{1,...n\}$, so dass der Knoten p der Vater von j ist, genau dann, wenn

$$p = min\{i | i > j, l_{ij} \neq 0\},$$

wobei l_{ij} die Koeffizienten des Cholesky-Faktors L von A sind. Ist A irreduzibel, dann definiert die Vater-Sohn-Beziehung einen Baum. Des weiteren bezieht sich j sowohl auf die Spalte einer Matrix, als auch auf den dazugehörigen Knoten in dem Eliminations-Baum.
Im Folgenden wird die Notation $T(A)$ benutzt, um den Eliminations-Baum von A darzustellen. Besteht keine Verwechslungsgefahr, wird T als Abkürzung für $T(A)$ benutzt. Mit der Struktur der Faktorisierungsspalte ist die Menge der Zeilenindizes der Nicht-Null-Elemente in der Spalte gemeint.

Satz 4.2.1. *Wenn der Knoten k ein Nachkomme von j im Eliminations-Baum ist, dann ist die Struktur des Vektors* $(l_{jk}, ..., l_{nk})^t$ *in der Struktur von* $(l_{jj}, ..., l_{nj})^t$ *enthalten.*
Satz 4.2.2. *Ist* $l_{jk} \neq 0$ *und* $k < j$*, dann ist im Eliminations-Baum der Knoten* k *ein Nachkomme von* j*.*

Die Notation $T[j]$ wird verwendet, um die Menge der Nachkommen des Knotens j im Eliminations-Baum T einschließlich j darzustellen. Mit anderen Worten: j und die Menge der Subbäume, die in dem Knoten j verwurzelt sind, sind enthalten.
Bei der Faktorisierung bzw. der Vorwärts-Substitution wird der Baum von den Blättern zur Wurzel, bei der Rückwärts-Substitution von der Wurzel zu den Blättern hin durchlaufen. Die einzige Anforderung, die erfüllt sein muss, ist, dass ein Sohn seine Eliminierungsoperationen beendet haben muss, bevor der Vater vollständig bearbeitet werden kann. Daher definiert der Eliminations-Baum nur eine partielle Ordnung für die Faktorisierung. Somit kann die Parallelität im Baum ausgenutzt werden (Baumparallelismus).

4.2.2 Faktorisierung

Für die Faktorisierung wird das Cholesky-Verfahren verwendet. Dabei ist A die gegebene $n \times n$-Matrix und L der dazugehörige Cholesky-Faktor. Man betrachte die j-te Spalte von L. Seien $i_0, i_1, ..., i_r$ die Zeilenindizes der Nicht-Null-Elemente von L_{*j} mit $i_0 = j$; das heißt, dass die j-te Spalte r Nicht-Null-Elemente unterhalb der Diagonalen besitzt.

Bei der Cholesky-Zerlegung wird wie folgt vorgegangen:
Für alle $j = 1, ...n$

Seien $j, i_1, ..., i_r$ die Stellen der Nicht-Null-Elemente.
Seien $c_1, ..., c_s$ die Söhne von j im Eliminations-Baum.

$$F_j = \begin{pmatrix} a_{j,j} & a_{j,i_1} & \dots & a_{j,i_r} \\ a_{i_1,j} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{i_r,j} & 0 & \dots & 0 \end{pmatrix} \dotplus U_{c_1} \dotplus ... \dotplus U_{c_s}, \tag{4.2}$$

wobei F_j die Frontal-Matrix und $U_{c_1} \dotplus ... \dotplus U_{c_s}$ die Update-Matrix ist. Für diese Frontal-Matrix wird nun die Cholesky-Zerlegung der ersten Spalte durchgeführt.

$$F_j = \begin{pmatrix} l_{j,j} & 0 \\ l_{i_1,j} & \\ \vdots & I \\ l_{i_r,j} & \end{pmatrix} \begin{pmatrix} 1 & 0 \\ & \\ & \\ 0 & U_j \end{pmatrix} \begin{pmatrix} l_{j,j} & l_{i_1,j} & \dots & l_{i_r,j} \\ 0 & & I & \end{pmatrix}.$$

Dabei erhält man die j-te Spalte der Matrix L und die Update-Matrix U_j. U_j enthält die Einträge der Söhne, wobei Söhne immer in vorhergehenden Schritten bestimmt werden, da der Baum von unten nach oben durchlaufen wird.

Zu beachten ist, dass immer nur die direkten Söhne betrachtet werden, d.h. die Söhne, die genau ein Level tiefer als der Vaterknoten liegen.
Eine Optimierung des Algorithmus kann erreicht werden, indem der Baum optimal durchlaufen wird. Dann können die Söhne U_j auf einen Stack gelegt werden und für jeden Schritt liegen dann genau diejenigen Söhne als erstes Element auf dem Stack, die auch gebraucht werden.

4.2.3 Lösen

Zuletzt wird das Gleichungssystem, auf dem die beiden vorherigen Schritte durchgeführt wurden, mit Rückwärts- und Vorwärts-Substitution gelöst. Dabei müssen die Vektoren der rechten Seiten, die zu einer Matrix zusammengefasst sind, vom Master per Broadcast den anderen Prozessoren übermittelt werden. Es wird auf die Update-Matrizen zurückgegriffen und die Baumstruktur ausgenutzt.

4.3 Erweiterung

In diesem Kapitel wird beschrieben, wie die Faktorisierung parallelisiert wird und wie eine Verbesserung der Laufzeit erreicht werden kann. Außerdem werden einige Sonderfälle besprochen.

4.3.1 Approximate-Minimum-Degree-Ordering

Um die Laufzeit zu reduzieren, wird in der Analysephase vor dem symbolischen Faktorisieren ein Ordering durchgeführt.
Beim Minimum-Degree-Algorithmus wird der Eliminations-Baum von den Blättern bis zur Wurzel aufgebaut. Es wird eine Permutationsmatrix P gesucht, so dass die Cholesky-Faktorisierung $PAP^T = LL^T$ weniger Nicht-Null-Einträge hat, als die Cholesky-Faktorisierung von A enthält [16].

4.3.2 Statische Abbildung

Die Abbildung, oder auch das Mapping, ist das Verteilen des Eliminations-Baumes auf die Prozessoren. Die Verteilung wird statisch durchgeführt und ist Teil der Analysephase. Die Hauptziele dieser Phase sind die Kontrolle der Kommunikationskosten sowie der Ausgleich des benutzten Speichers und der Rechenkosten auf den Prozessoren. Die Rechenkosten werden durch die Anzahl der Floating-Point-Operationen angenähert.
In diesem Abschnitt wird der Algorithmus der statischen Abbildung des Eliminations-Baumes auf die Prozessoren beschrieben. Des weiteren wird gezeigt, wie Speicher- und Arbeitsausgleichskriterien kombiniert werden.

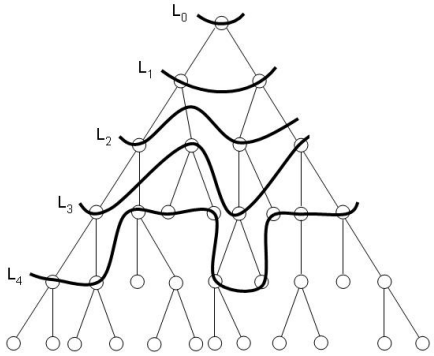

Abbildung 4.2: Zerlegung des Eliminations-Baumes in die jeweiligen Levels

Dabei wird der Baum Level für Level (Abbildung 4.2) von unten nach oben abgearbeitet. Das Level L_0 wird, wie im Algorithmus (Abbildung 4.3) und der Abbildung 4.4 beschrieben, ermittelt.

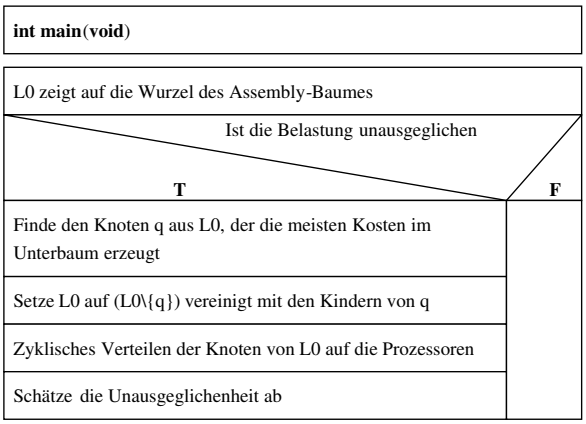

Abbildung 4.3: Konstruktion und Abbilden des Anfangs-Levels L_0

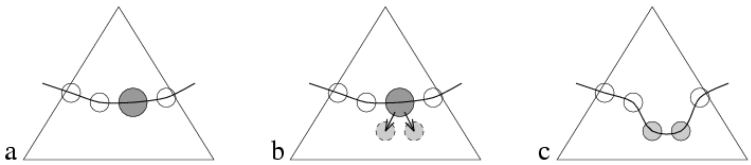

Abbildung 4.4: Ein Schritt in der Konstruktion des ersten Levels L_0

Ein Knoten gehört zu L_i, $i > 0$, genau dann, wenn alle seine Söhne zu L_j, $j <= i - 1$, gehören. Als erstes werden die Knoten des Levels L_0 (und die dazugehörenden Unterbäume) abgebildet. Dieser erste Schritt wird durchgeführt, um die Arbeit in den Unterbäumen auszugleichen und um die Kommunikation zu reduzieren, da alle Knoten eines Unterbaumes auf den gleichen Prozessor abgebildet werden. Normalerweise ist es nötig viel mehr Knoten auf dem Level L_0 zu haben, als Prozessoren zur Verfügung stehen, um einen Ausgleich zu erhalten. Daher ist L_0 abhängig von der Anzahl der Prozessoren, denn durch eine größere Anzahl an Prozessoren erhält man kleinere Unterbäume, wenn die Matrixgröße dies noch zulässt.
Das statische Abbilden höherer Levels bewirkt nur einen Ausgleich des Speicherplatzes. Als erstes wird für jeden Prozessor der Speicherplatzbedarf für die Knoten des Levels L_0 berechnet. Für jedes weitere Level $L_i, i > 0$, wird jeder nicht abgebildete Knoten von L_i auf den Prozessor mit dem kleinsten Speicherplatzbedarf abgebildet.
Danach wird das statische Abbilden verwendet, um explizit die permutierte Ausgangsmatrix auf die Prozessoren zu verteilen, und um den Umfang der Arbeit und den Speicherplatzbedarf auf jedem Prozessor abzuschätzen.

4.3.3 Parallelisierung und ihre Ergebnisse

Bei der Parallelisierung wird zwischen drei Typen unterschieden:

- **Typ1-Parallelismus**: Bei der Typ1-Parallelisierung liegt der serielle Fall vor. Die Frontal-Matrix F wird auf einem Prozessor berechnet.

- **Typ2-Parallelismus**: Im Fall der Typ2 Parallelisierung liegt eine große Frontal-Matrix F vor. Die Cholesky-Faktorisierung wird in mehrere Teile unterteilt, indem F spaltenweise auf n Prozessoren verteilt wird. Diese Parallelisierung wird auch als *1D-Block-Partitionierung* bezeichnet.

- **Typ3-Parallelismus**: Bei diesem Typ des Parallelismus wird die Frontal-Matrix in Blöcke unterteilt. Diese Blöcke sind aber, im Gegensatz zum Typ2, nicht nur Spalten- sondern echte Teil-Matrizen. Ein Block kann beispielsweise die Elemente der ersten bis dritten Spalte und Zeile enthalten (siehe Abbildung 4.5). Diese Unterteilung wird als 2D-Block-Partitionierung bezeichnet.

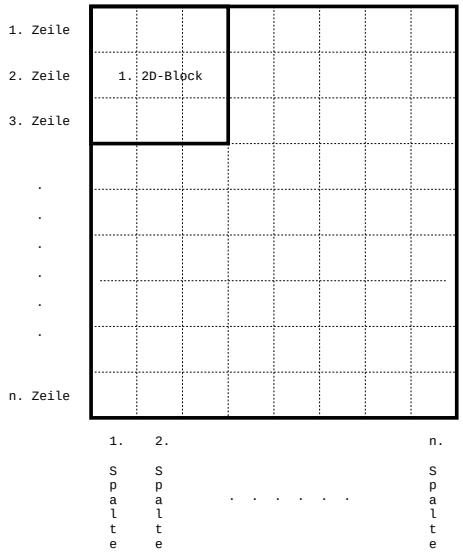

Abbildung 4.5: 2D-Block-Partitionierung

Die Unterteilung in diese verschiedenen Typen des Parallelismus wird vorgenommen, da in den oberen drei Levels der größte Teil der Berechnungen, etwa 75 Prozent der Gesamtberechnung, stattfindet. Die Knoten, die im unteren Bereich des Baumes liegen, werden nur auf einen Prozessor gelegt (Typ1-Parallelisierung). Werden die Blöcke der Frontal-Matrix, die auf einem Knoten liegen größer, wird der Typ2, und in dem der Wurzel am nächsten liegenden Knoten wird der Typ3-Parallelismus angewandt. Wie dies aussehen kann, ist in Abbildung 4.6 dargestellt.

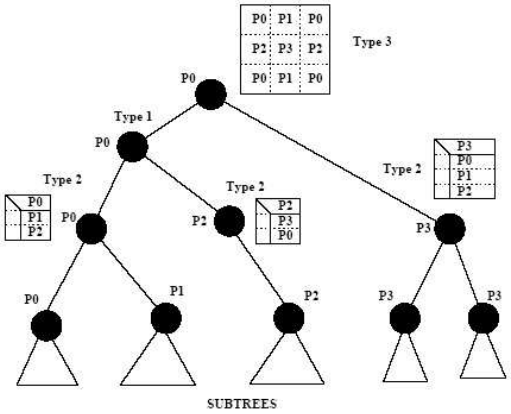

Abbildung 4.6: Aufteilen des Eliminations-Baumes in die drei Typen des Parallelismus

Beim Rechnen auf mehreren Prozessoren ist Kommunikation erforderlich (Kapitel 2). Um Laufzeit zu sparen, wird asynchrone Kommunikation benutzt. Das heißt, dass Berechnung und Kommunikation parallel stattfinden können. Dabei wird vor der Faktorisierung eine Abschätzung vorgenommen, wie groß der Puffer für die zu versendenden und die zu empfangenden Nachrichten sein muss. Die Abschätzung erfolgt anhand der Parallelismus-Typen. Nach der Abschätzung wird der Puffer auf jedem Prozessor angelegt.
Zu Beginn der Berechnungen wird statisches Scheduling vorgenommen. Im Laufe der Berechnun-

gen kann es wegen der ungleichmäßigen Verteilung im Eliminations-Baum dazu kommen, dass ein Prozessor noch Berechnungen durchführen muss, während die anderen fertig sind und warten müssen. An dieser Stelle wird auf dynamisches Scheduling umgeschaltet.

4.3.4 Assembly-Prozess

Eine Abschätzung der Struktur der Frontal-Matrix (Größe, Anzahl der vollständig summierten Variablen), wird während der Analysephase durchgeführt. Die endgültige Struktur und die Liste der Indizes in der Frontal-Matrix wird während des Assembly-Prozesses der Faktorisierungsphase bestimmt. Die Liste der Indizes einer Frontal-Matrix besteht aus der Verknüpfung der Indexlisten der Contribution-Blöcke der Söhne und des Arrowheads, (Pfeilspitze) d.h. des Koeffizienten in der ersten Spalte bzw. Zeile der Frontal-Matrix (vergleiche 4.2). Die Liste wird anschließend mit allen vollständig summierten Variablen der Frontal-Matrix verknüpft. Sobald die Indexliste der Frontal-Matrix berechnet wurde, kann das Assemblieren der numerischen Werte effizient durchgeführt werden.
Sei nun *Inode* ein Knoten mit Typ 2-Parallelismus. Der Master des *Inodes* bestimmt die Partition in die unterschiedlichen Zeilenblöcke der Frontal-Matrix und wählt eine Menge von Slave-Prozessoren aus, die an der parallelen Assemblierung/Faktorisierung des *Inodes* teilnehmen. Der Master sendet für den *Inode* eine Nachricht (zum Beispiel mit dem tag DESC_STRIP) an jeden Slave, die die Arbeit beschreibt, die jeder Slave ausführen muss. Er sendet ebenfalls eine Nachricht (mit dem tag MAPROW) an alle Slave-Prozessoren seiner Söhne, um denen mitzuteilen, wohin die Contribution-Blöcke für den Assembly-Prozess gesendet werden sollen. Dabei ist tag eine Variable, normalerweise vom Typ Integer, mit der eine Nachricht eindeutig identifiziert werden kann.
Die Reihenfolge der gesendeten Nachrichten ist von großer Bedeutung: ein Slave des *Inodes* muss zuerst einen Contribution-Block empfangen, bevor er die Nachricht des tags DESC_STRIP von seinem Master erhält. Um diesem Slave einen blockierten Empfang der vermissten Nachricht DESC_STRIP zu erlauben, muss sichergestellt sein, dass der Master-Knoten zunächst den DESC_STRIP gesendet hat, bevor er den MAPROW sendet. Andererseits kann nicht garantiert werden, dass DESC_STRIP tatsächlich gesendet wird (der Grund dafür kann ein voller Sendepuffer sein).
Für die Effizienz ist globales Ordering der Indizes in der Frontal-Matrix nötig, um garantieren zu können, dass alle unteren Dreieckseinträge in einer Contribution-Reihe eines Sohnes zu der entsprechenden Reihe im Vaterknoten dazugehören. Dieses globale Ordering erhält man in der Analysephase. Die Reihenfolge entspricht dabei derselben, wie die Elimination der Variablen, wenn kein numerisches Pivoting auftritt.
Häufig reicht es aus, nur den Arrowhead, verknüpft mit den ersten vollständig summierten Variablen jeder Frontal-Matrix, zu sortieren. Somit kann der Assembly-Prozess für die Liste der Knotenindizes, wie in Abbildung 4.7 beschrieben, durchgeführt werden.

int main(**void**)
1. Merge-Sort der sortierten Listen der Indizes der Söhne und des ersten Arrowheads
2. Bilden und Sortieren der Variablen, die nur zu den anderen Arrowheads dazugehören (und im ersten Schritt nicht gefunden worden)
3. Zusammenfassen der Listen, die im 2. Schritt gebildet wurden, mit den sortierten Listen, die man im ersten Schritt erhalten hat

Abbildung 4.7: Algorithmus zum Assemblieren von Indizes in einem Vaterknoten

4.3.5 Anmerkungen

In dieser Arbeit werden nur symmetrisch positiv definite Matrizen verwendet und untersucht. Daher eine kleine Anmerkung für allgemeinere Fälle :

- Bei irreduziblen Matrizen wurde bisher die Datenstruktur Baum verwendet. Ist die Matrix reduzibel, dann handelt es sich bei der Datenstruktur um einen Wald. Dieses System kann in mehrere kleinere irreduzible Systeme zerlegt werden, wobei jede irreduzible Komponente einen Baum darstellt. Jede dieser Baum-Komponenten des Waldes wird dann separat durchlaufen.
- Im Fall einer asymmetrischen Matrix wird, anstatt der hier beschriebenen Cholesky-Zerlegung, die LU-Zerlegung angewandt.
- Je tiefer der Eliminations-Baum ist, desto dichter ist die Matrix besetzt. Jeder Knoten des Eliminations-Baumes entspricht den Operationen bei der Zerlegung einer vollbesetzten Untermatrix, der sogenannten Frontal-Matrix.

Kapitel 5

Auswertung

5.1 Motivation

Mit den in Kapitel 3 beschriebenen Tools für Zeitmessung und graphische Darstellung hat man Ergebnisse erhalten, mit denen nun die eigentlichen Performance-Untersuchungen durchgeführt werden können.
Es werden jeweils die Hauptfunktionen der fünf unterschiedlichen Gleichungssystemlöser betrachtet [6]:

- *Cholesky_1*: Vor der eigentlichen Zerlegung der Gleichungssysteme werden eine Initialisierung, eine Anordnung und eine symbolische Faktorisierung der Eingabematrix vorgenommen.
- LDL^T_1: Es wird mit den gleichen Schritten vor der eigentlichen Zerlegung wie bei *Cholesky_1* vorgegangen.
- *Cholesky_2*: Der Unterschied zwischen *Cholesky_1* und *Cholesky_2* besteht darin, dass bei *Cholesky_2* die Eingabematrix nicht permutiert wird, sondern dass während der Faktorisierung die Permutation durchgeführt wird.
- LDL^T_2: Es wird ebenfalls, wie bei *Cholesky_2*, mit nichtpermutierter Eingangsmatrix gearbeitet.
- *Expert Driver*: Kombiniert die LDL^T-Zerlegung, das Lösen der Dreieckssysteme und die Nachiteration.

Die Messungen wurden abhängig von der Problemgröße (Matrix-Dimension n bzw. Grad der Besetztheit) und der Anzahl der Prozessoren p durchgeführt, wobei sowohl Threads-basiert unter Verwendung des gemeinsamen Speichers als auch MPI-basiert parallel gerechnet wurde. Anschließend wurden Performance-Untersuchungen für die Thread- und MPI-basierten Messungen durchgeführt und miteinander verglichen. Außerdem wurden die Auswirkung der Anzahl der rechten Seiten (Rhs) auf die Rechenzeit überprüft.

5.2 Amdahls Gesetz

Beim Parallelisieren eines Programms wird die Arbeit auf p Prozessoren verteilt. Dabei erhöht sich die Leistung im Vergleich zur Rechnung auf einem Prozessor aber nicht auf das p-fache, da sonst immer alle Prozessoren gleichzeitig arbeiten müssten. In jedem parallelen Programm gibt es sequentielle Anteile, wie z.B. die Datenverteilung. Befindet sich das Programm in so einem sequentiellen Abschnitt arbeitet nur ein Prozessor. Die übrigen $p-1$ Prozessoren werden dabei nicht in Anspruch genommen. Die maximale Leistung kann also nicht erreicht werden. Der sequentielle Teil wird mit dem Gesetz von Amdahl bestimmt. Je höher dieser ist, desto weniger lohnt sich die

Parallelisierung. Im Folgenden wird der sequentielle Teil mit f bezeichnet.
Definition 5.2.1. *Sei $T(1)$ die Programmbearbeitungszeit auf einem Einprozessorsystem und $T(p)$ die benötigte Zeit auf p Prozessoren. Der Zeitgewinn durch die Parallelverarbeitung, der Speedup (siehe auch Kapitel 5.3.5), wird berechnet mit*

$$S(p) = T(1)/T(p).$$

Da parallele Programme auch sequentielle Teile enthalten, wird die Bearbeitungszeit in einen parallelen- T_p und sequentiellen- T_s Teil zerlegt. Somit berechnet sich die Programmbearbeitungszeit auf einem Einprozessorsystem folgendermaßen:

$$T(1) = T_s + T_p$$

Sei nun $f = \frac{T_s}{T_s+T_p}$, für $0 \leq f \leq 1$ der sequentielle Anteil des Programms, dann gilt

$$T(p) = f \cdot T(1) + \frac{(1-f) \cdot T(1)}{p} = T_s + \frac{T_p}{p}.$$

Mit diesem Ansatz erhält man nun den Speedup

$$S(p) = \frac{T(1)}{T(p)} = \frac{p}{1 + f \cdot (p-1)} = \frac{1}{f + \frac{1-f}{p}}.$$

Mit Hilfe dieses Speedups kann nun der sequentielle Teil der Messergebnisse berechnet werden. Außerdem sieht man, dass der Speedup $S(p)$ nie größer als $\frac{1}{f}$ werden kann, egal wie groß p (Zahl der Prozessoren) gewählt wird [1].

5.3 Performance der Shared Memory Version mit Threads

Zuerst soll die Performance für die Threads-basierte Shared Memory Version der Bibliothek WSMP getestet werden. Dabei wurden Zeitmessungen mit bis zu 32 Prozessoren durchgeführt. Dies ist auch die maximale Anzahl an Prozessoren, mit der auf einem Knoten gerechnet werden kann und mit der Shared Memory parallele Programme Threads- oder OpenMP-basiert durchgeführt werden können.

5.3.1 Anzahl der rechten Seiten

Die Anzahl der rechten Seiten sollte keine Auswirkung auf die Rechenzeit haben, da die folgenden Messergebnisse in Tabellen und Graphiken nur die Zeiten für die eigentliche Faktorisierung enthalten. Dies ist nur bei dem Expert Driver nicht der Fall.
Trotzdem wurde die Auswirkung der Anzahl der rechten Seiten auf die Rechenzeit untersucht. Alle Messungen wurden mit je einer und zehn rechten Seiten durchgeführt. Dabei ist, wie vermutet, zu erkennen, dass die Anzahl der rechten Seiten (siehe Tabelle 5.1 und 5.2) kaum eine Auswirkung auf die Rechenzeit hat. Allerdings ist bei der *Cholesky_1*- und LDL^T_1-Zerlegung ein Unterschied zu erkennen. Die Rechenzeit verdoppelt sich bei diesen Verfahren nahezu. Der zusätzliche Rechenaufwand bei *Cholesky_1*- und LDL^T_1 lässt auf einen Fehler in der Bibliothek für diese Verfahren schließen. Allerdings ist dieser größere Rechenaufwand bei 32 Prozessoren nicht mehr zu erkennen. Hier scheint der Fehler nicht aufzutreten. Der Mehraufwand sollte nur in der Lösungsphase stattfinden, wobei diese im Vergleich zum Gesamtrechenaufwand nur einen kleinen Anteil an der gesamten Rechenzeit in Anspruch nimmt.

Erwähnenswert ist noch, dass das Programm mit einem Fehler beendet wurde, als manche rechte Seiten mit unterschiedlichen Dimensionen kombiniert wurden. Dieser Fehler trat beispielsweise

Laufzeiten der Verfahren in sec						
Dimension	Rhs	*Cholesky_1*	LDL^T_1	*Cholesky_2*	LDL^T_2	*Expert Driver*
132651	1	5.116	4.736	5.591	5.074	5.953
132651	10	7.016	6.408	5.621	5.073	6.388
226981	1	13.626	12.976	16.979	16.445	17.495
226981	10	18.244	17.591	16.811	16.272	18.804
357911	1	32.119	32.870	37.107	36.992	39.451
357911	10	53.879	53.244	37.555	37.148	40.646
531441	1	80.453	77.569	84.781	82.114	86.587
531441	10	186.104	177.799	84.160	82.529	89.981
753571	1	160.499	158.630	185.889	183.014	192.430
753571	10	254.074	249.981	186.195	184.048	195.873
1030301	1	285.571	284.074	269.919	272.848	283.675
1030301	10	484.821	490.722	271.716	270.835	288.669

Tabelle 5.1: Laufzeiten, p = 16, eine und zehn rechte Seiten

Laufzeiten der Verfahren in sec						
# Prozessoren	Rhs	*Cholesky_1*	LDL^T_1	*Cholesky_2*	LDL^T_2	*Expert Driver*
1	1	269.522	297.662	290.717	326.349	337.284
1	10	436.063	490.810	290.414	325.830	340.087
2	1	116.548	130.328	125.045	134.175	138.531
2	10	203.774	229.777	125.624	134.482	140.363
4	1	67.441	74.832	80.809	80.725	86.985
4	10	113.791	123.704	80.875	82.430	88.612
8	1	34.860	35.686	37.998	38.699	41.963
8	10	56.143	58.549	37.946	39.895	43.931
16	1	32.119	32.870	37.107	36.992	39.451
16	10	53.879	53.244	37.555	37.148	40.646
32	1	53.590	52.364	34.625	33.656	36.163
32	10	53.560	52.510	34.327	33.596	37.794

Tabelle 5.2: Laufzeiten, n = 357.911, eine und zehn rechte Seiten

bei vier rechten Seiten auf. Er entsteht durch die Bibliothek WSMP, da dieser beim Testen mit der neuen Version 4.8.5 nicht mehr aufgetreten ist. Dies könnte auch die abweichenden Zeiten bei zehn rechten Seiten erklären. Der Fehler bei der hier verwendeten Version 1.9.4 basiert vermutlich auf Speicherproblemen, die bei einigen Kombinationen von Matrixdimensionen und Anzahlen rechter Seiten auftreten.

5.3.2 Problemgröße

Für die Performance-Analyse ist die Änderung der Rechenzeit abhängig von der Dimension n der Matrix von Interesse. Hierbei wurden unterschiedliche Datensätze verwendet. Zum einen die Harwell-Boeing-Matrizen und zum anderen die anhand des Hexe27-Elementes erstellten dünnbesetzten Matrizen. Die zuletzt genannten Matrizen werden verwendet, da die verfügbaren Harwell-Boeing-Matrizen nicht sehr groß sind. Beide Datensätze werden im Anhang A beschrieben.
Die Harwell-Boeing-Matrizen standen mit Dimensionen von 4.884 bis 90.449 zur Verfügung. Bei den selbsterstellten Matrizen hingegen wurden Zeitmessungen für Größen von 132.651 bis 1.030.301 durchgeführt. Da die beiden Arten der Matrizen auf Grund des unterschiedlichen Grades der Besetztheit nicht miteinander verglichen werden können, werden im Folgenden nur die selbsterstellten

Matrizen dargestellt. Die Ergebnisse für die Harwell-Boeing-Matrizen findet man im Anhang B. Betrachtet man die Rechenzeit der verschiedenen Verfahren in Abhängigkeit von der Dimension, fällt auf, dass die Rechenzeiten der unterschiedlichen Routinen mit einer rechten Seite nicht bedeutend voneinander abweichen (Abbildung 5.1).

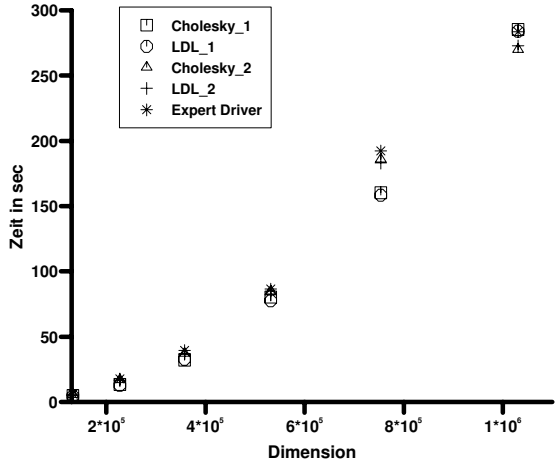

Abbildung 5.1: Vergleich der Verfahren, p = 16, eine rechte Seite

Allerdings sind bei zehn rechten Seiten *Cholesky_1* und LDL^T*_1* deutlich langsamer als die übrigen Verfahren (Abbildung 5.2).

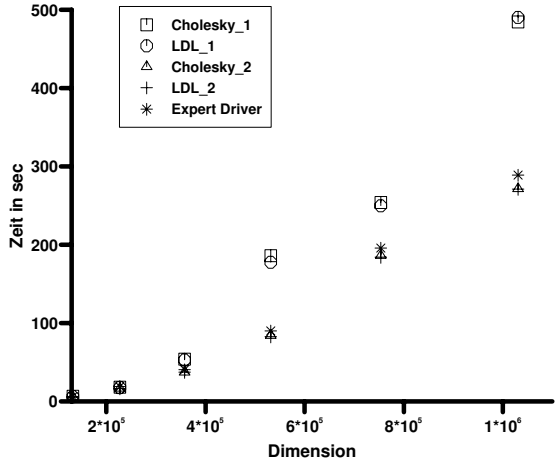

Abbildung 5.2: Vergleich der Verfahren, p = 16, zehn rechte Seiten

Zusätzlich fällt auf, dass die Rechenzeiten beider Verfahren (*Cholesky_1* und LDL^T_1) bei 32 Prozessoren keinen Unterschied mehr aufweisen, egal ob mit einer oder zehn rechten Seiten gerechnet wird. Die Rechenzeiten sind trotz allem noch deutlich höher als bei den anderen Verfahren. Hier scheint der Fehler der Bibliothek auch bei einer rechten Seite aufzutreten.

5.3.3 Anpassen der Laufzeiten

Zu den Messwerten in Tabelle 5.1 soll eine Funktion mit Hilfe eines nichtlinearen least-squares-Fit (Methode der kleinsten Fehlerquadrate) bestimmt werden. Dabei werden die quadratischen Abstände der Messpunkte zu einer Kurve minimiert.
Als Ansatzfunktion wird

$$g(n) = a \cdot n^b$$

gewählt, da für das Cholesky-Verfahren mit dichtbesetzten Matrizen der Fit $\frac{1}{3} \cdot n^3$ ist. Anhand der Abbildungen 5.1 und 5.2 ist zu erkennen, dass dies auch durchaus sinnvoll erscheint. $g(n)$ und n sind der Messwert und die Matrixdimension. Es sollen nun die Parameter a und b bestimmt werden. Mit Hilfe von Maple (Kapitel 3.5.2) wird die Summe

$$\sum_{i=1}^{m} (a \cdot n[i]^b - y[i])^2$$

über die m Messwerte $y[i]$ minimiert, das heißt, es werden die Ableitungen nach a und b gebildet und diese gleich Null gesetzt. Durch das Lösen des nichtlinearen Gleichungssystems erhält man schließlich die Parameter a und b (siehe Tabelle 5.3).

Verfahren	a	b
Cholesky_1	0.6385098756e-9	1.937978020
LDL^T_1	0.4752868356e-9	1.958893330
Cholesky_2	0.2114020513e-7	1.682933710
LDL^T_2	0.1175978322e-7	1.725798544
Expert Driver	0.1644650757e-7	1.704471705

Tabelle 5.3: Fit-Parameter der verschiedenen Verfahren, p = 16

Wie in Tabelle 5.3 zu sehen, sind die Werte der Parameter für die verschiedenen Verfahren ähnlich. Die gefittete Funktion

$$g(n) = 0.2114020513 \cdot 10^{-7} \cdot n^{1.682933710}.$$

für *Cholesky_2* ist in Abbildung 5.3 dargestellt. Demnach skaliert die Rechenzeit, da die Matrizen dünnbesetzt sind, etwa mit n^2, im Gegensatz zu n^3 wie bei vollbesetzten Matrizen.

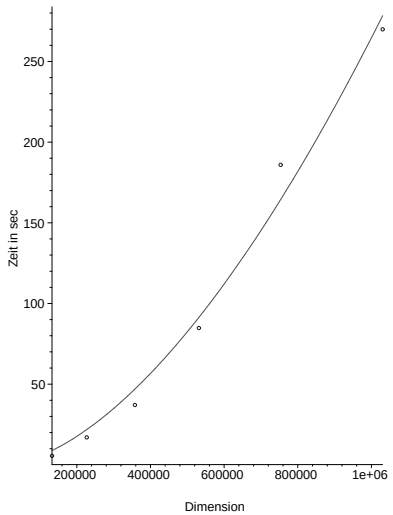

Abbildung 5.3: Fit der Ausführungszeiten für *Cholesky_2*, p = 16

5.3.4 Flip-Rate

Mit Hilfe des HPM-Tools (Kapitel 3.4.2) wurden die *Flip*-Raten (Floating Point Instructions plus FMAs rate (Mflip/sec)) [8] für die unterschiedlichen Verfahren gemessen. Diese entsprechen, wie bereits erwähnt, Millionen Floating Point Operations per Second (MFLOPs).
Die *Flip*-Rate ist bei allen Verfahren ziemlich identisch und unabhängig von der Dimension der Matrix.

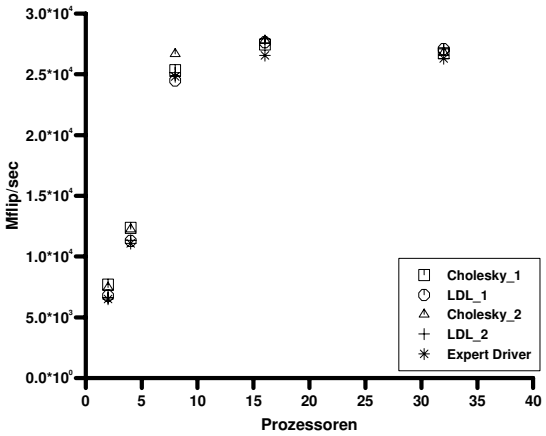

Abbildung 5.4: Gesamt-*Flip*-Rate in Abhängigkeit von p, $n = 1.030.301$

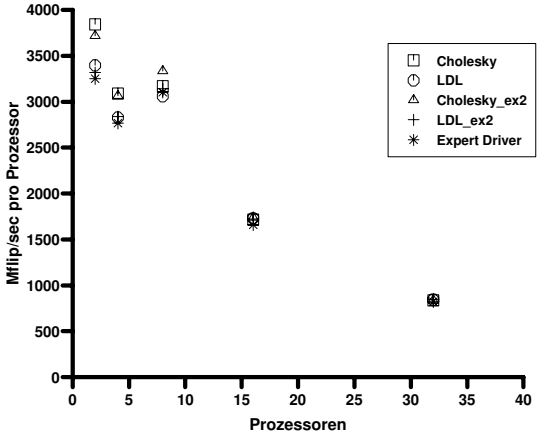

Abbildung 5.5: *Flip*-Rate pro Prozessor in Abhängigkeit von p, $n = 1.030.301$

Wie in Abbildung 5.5 erkennbar, nimmt die *Flip*-Rate pro Prozessor bis acht Prozessoren nur schwach ab. Danach ist ein starker Abfall zu erkennen. Der Grund dafür ist, dass auf dem IBM p690 (Kapitel 2.3) bei Shared Memory Parallelisierung mit Threads ab einer bestimmten Prozessoranzahl der Speicher nicht mehr ausreichend ausgenutzt werden kann. Die Speicherzugriffskonflikte nehmen zu und es kann keine höhere *Flip*-Rate mehr erreicht werden. Dieses Verhalten geht so weit, dass bei 32 Prozessoren (siehe auch Tabelle 5.4) die Rate sogar abfällt. Die maximale Anzahl von 32 Prozessoren kann also nicht gewinnbringend ausgenutzt werden.

Flip-Rate in Mflip/sec					
# Prozessoren	*Cholesky_1*	LDL^T_1	*Cholesky_2*	LDL^T_2	*Expert Driver*
1	3788.231	3640.809	3813.030	3633.593	3531.832
2	7185.232	6961.580	7073.080	7239.243	6487.108
4	11745.913	11238.488	12432.595	11578.366	12163.480
8	23832.020	24548.718	23678.892	24100.063	21612.642
16	25074.685	26336.195	24772.735	25582.998	24167.313
32	25349.859	26458.440	23332.237	25524.886	22675.627

Tabelle 5.4: Gesamt-Flip-Raten der verschiedenen Verfahren, n = 226.981

5.3.5 Speedup

Um die Effizienz der vorgenommenen Parallelisierung der Berechnungen beurteilen zu können, wird der Speedup mit folgender Formel berechnet [17]:

$$S(p) = \frac{T(1)}{T(p)} \approx \frac{2T(2)}{T(p)}.$$

Der Speedup bezeichnet das Verhältnis der Rechenzeit $T(1)$ einer Anwendung auf einem Einprozessorsystem zur Rechenzeit $T(p)$ auf einem Parallelrechner mit p Prozessoren. Ist das Programm

nicht auf einem Prozessor ausführbar, kann der Speedup mit $\frac{2T(2)}{T(p)}$ angenähert werden.

# Prozessoren	132.651	226.981	357.911	531.441	753.571	1.030.301
1	1	1	1	1	-	-
2	1.909	1.951	2.313	1.987	2	2
4	3.107	3.170	3.996	3.259	3.323	3.278
8	6.218	6.483	7.731	6.417	6.509	6.676
16	6.518	6.842	8.391	6.524	6.809	7.233
32	4.854	5.034	5.029	2.930	4.325	4.147

Tabelle 5.5: Speedup für unterschiedliche n für *Cholesky_1*

Wie man den bisherigen Ergebnissen zufolge vermuten kann, und auch in Tabelle 5.5 erkennbar ist, zeigt der Speedup, dass ab etwa acht Prozessoren die Auslastung des Parallelrechners nicht mehr gegeben ist.
Desweiteren ist in Abbildung 5.6 zu sehen, dass die *Cholesky_1*-Zerlegung die schlechtesten Speedup-Werte der fünf Verfahren liefert. Das bestätigt zusätzlich den schlechten Speedup der einzelnen Verfahren.

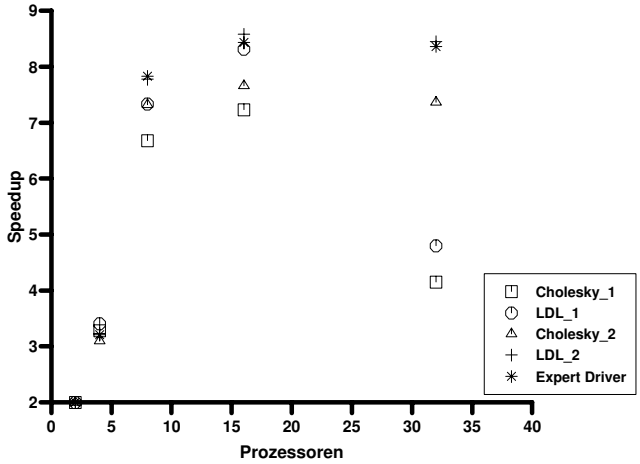

Abbildung 5.6: Speedup der verschiedenen Verfahren, n = 1.030.301

Normalerweise ist der maximale Speedup durch die Anzahl der Prozessoren, auf denen gerechnet wird, begrenzt. In der Tabelle 5.5 hingegen ist zu sehen, dass bei der Berechnung der Matrix mit der Dimension 357.911 auf zwei Prozessoren der Speedup

$$S(p) = 2.313 > 2 = p$$

ist. Damit übersteigt der Speedup die Anzahl der Prozessoren. Der Grund dafür könnte sein, dass bei der sequentiellen Ausführung der Speicherzugriff langsamer ist, da ein kleinerer Cache zur Verfügung steht.

5.3.6 Bestimmung des seriellen Anteils

Die Bestimmung des seriellen Anteils, also der Teil, der nicht parallelisiert werden konnte, wird mit dem in Kapitel 5.2 beschriebenen Gesetz von Amdahl berechnet. Bei der bisherigen Auswertung konnte festgestellt werden, dass ab etwa acht Prozessoren die Prozessoren nicht einmal annähernd ausgenutzt werden können. Daher ist auch eine Bestimmung des seriellen Anteils nicht sinnvoll.

5.4 Performance der MPI-Version

Bei der Performance-Analyse der MPI-Programme wurde mit 1, 2, 4, 8, 16 und 32 Prozessoren gerechnet. 32 Prozessoren ist auch die maximale Anzahl, die auf jedem Knoten zur Verfügung steht (siehe Abschnitt 2.3). Die Berechnungen wurden lediglich auf einem Knoten durchgeführt. Die dünnbesetzten, symmetrisch positiv definiten Matrizen wurden auf der Basis von Hexe27-Elementen erstellt (Anhang A). Dabei wurde die Erstellung der Matrix auf dem Masterknoten mit anschließendem Verteilen auf die Slaves und die direkt verteilte Generierung der Matrix auf den Prozessoren unterschieden. Letzteres wird im Folgenden mit *peer* bezeichnet.

5.4.1 Anzahl der rechten Seiten

Die Anzahl der rechten Seiten spielt bei der MPI-Version keine wesentliche Rolle. Hierbei sind nur minimale Abweichungen (Tabelle 5.6) zu erkennen.

Laufzeiten der Verfahren in sec						
# Prozessoren	Rhs	Cholesky_1	LDL^T_1	Cholesky_2	LDL^T_2	Expert Driver
1	1	268.046	299.334	290.848	323.893	326.254
1	10	267.939	295.521	290.375	326.161	335.115
2	1	124.582	134.444	127.149	136.293	135.594
2	10	125.153	133.733	126.539	133.726	136.221
4	1	74.270	76.905	75.382	75.708	78.721
4	10	74.419	77.494	73.527	76.862	80.786
8	1	37.537	38.249	38.806	39.705	42.347
8	10	41.529	37.918	39.273	38.931	45.165
16	1	24.581	26.293	26.434	23.986	24.250
16	10	23.405	26.007	26.940	23.013	25.783
32	1	13.142	12.326	13.208	12.314	12.662
32	10	13.105	12.283	13.026	12.422	13.423

Tabelle 5.6: Laufzeiten mit einer und zehn rechten Seiten, n = 357.911, nicht-*peer*

Tabelle 5.6 zeigt die Messwerte der nicht-*peer* Version. Die Werte der *peer* Version sehen ähnlich aus. Auch hier gibt es kaum Laufzeitunterschiede zwischen einer und zehn rechten Seiten.

5.4.2 Problemgröße

Bei dem Vergleich der unterschiedlichen Dimensionen n sind keine wesentlichen Auffälligkeiten zu erkennen. Alle Verfahren haben etwa die gleiche Laufzeit. Zusammenfassend sind die Cholesky-Verfahren etwas schneller als der Expert Driver (Abbildung 5.7).

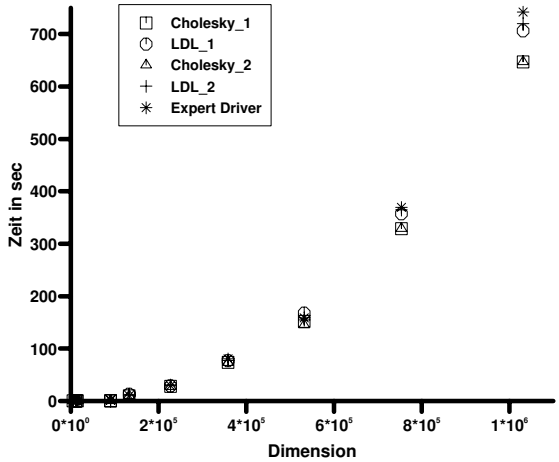

Abbildung 5.7: Laufzeiten der verschiedenen Verfahren, p = 4, zehn rechte Seiten

Auch hierbei gibt es keine auffälligen Unterschiede zwischen *peer* und nicht-*peer* (Tabelle 5.7).

Laufzeiten der Verfahren in sec						
Dimension	Cholesky_2	Cholesky_2 *peer*	LDL^T_2	LDL^T_2 *peer*	Expert Driver	Expert Driver *peer*
132651	10.727	12.459	10.035	11.016	10.145	10.068
226981	29.811	28.091	28.392	29.054	28.329	29.054
357911	75.382	74.699	75.710	77.695	78.721	77.695
531441	154.634	153.432	165.529	167.260	167.767	167.260
753571	327.709	328.767	357.815	357.654	358.515	357.654
1030301	643.477	644.135	714.114	713.368	715.196	713.368

Tabelle 5.7: Laufzeiten der Verfahren mit und ohne *peer*, p = 4, eine rechte Seite

Eine wesentliche Auffälligkeit trat bei der nicht-*peer*-Variante bei der Berechnung auf 32 Prozessoren auf. Bei der Erstellung der Matrix mit Hexe27-Elementen wurden die Programme bei Matrizen mit Dimensionen größer 531.441 mit einem Segmentation-fault beendet. Die *peer*-Variante hingegen lief weder mit Dimensionen von 132.651 bis 226.981 noch mit Dimensionen von 753.571 bis 1.030.103. Bei der Berechnung auf 16 Prozessoren liefen die Programme mit der Matrixdimension 1.030.301 lediglich bei der *peer*-Variante bei den Verfahren *Cholesky_2* und LDL^T_2. Dieses Problem basiert auf einem Fehler in der Bibliothek WSMP und konnte nicht weiter untersucht werden, da auf den Quellcode der Bibliothek nicht zugegriffen werden kann. Aus diesem Grund konnten bei der MPI-Version nicht dieselben Probelmgrößen dargestellt werden wie bei der Threads-Version.
Die Berechnungen mit der Harwell-Boeing-Matrix **BCSSTM25**, bei der nur die Diagonalelemente ungleich Null sind (siehe Anhang A), laufen nur auf einem Prozessor. Dies kommt daher, dass zu wenig Elemente ungleich Null sind. Daher ist im Programm die Möglichkeit der Verteilung nicht vorgesehen.

Im Weiteren wird nur noch die Erstellung der Matrix auf dem Master (nicht-*peer*-Variante) betrachtet, da keine bemerkenswerten Unterschiede zwischen der *peer*- und nicht-*peer*-Variante festgestellt werden konnten.

5.4.3 Anpassen der Laufzeiten

Für die MPI-Parallelisierung wird, ebenso wie bei Verwendung von Threads, mit der Methode der kleinsten Fehlerquadrate ein Fit für die gemessenen Ausführungszeiten in Abhängigkeit von der Matrixdimension berechnet. Auch hier wird als Ansatzfunktion

$$g(n) = a \cdot n^b$$

verwendet. Es werden schließlich die folgenden Werte für die Parameter a und b berechnet (siehe Tabelle 5.8).

Verfahren	a	b
Cholesky_1	0.1525131337e-9	2.012757434
LDL^T_1	0.2854903078e-9	1.969339716
Cholesky_2	0.7619991283e-9	1.895290185
LDL^T_2	0.818100969e-10	2.062141961
Expert Driver	0.7619991283e-9	1.895290185

Tabelle 5.8: Fit-Parameter der verschiedenen Verfahren, p = 16

Auch hier sind die Graphen der verschiedenen Verfahren vergleichbar. Die Messpunkte können durch den jeweiligen Graphen sehr gut angenähert werden. Abbildung 5.8 zeigt als Beispiel den Fit der Messergebnisse des *Cholesky_2*-Verfahrens. Es wird hierbei die Funktion

$$f(n) = 0.7619991283 \cdot 10^{-9} \cdot n^{1.895290185},$$

mit den Parametern $a = 0.7619991283 \cdot 10^{-9}$ und $b = 1.895290185$ dargestellt.

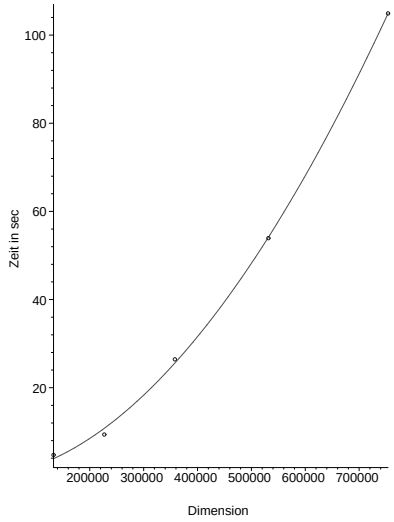

Abbildung 5.8: Fit der Ausführungszeiten für *Cholesky_2*, p = 16

Auch bei der MPI-Version ist also für Hexe-27-Matrizen die n^2 -Skalierung erkennbar.

5.4.4 Flip-Rate

Bei Verwendung von MPI wird von dem HPM-Tool (Kapitel 3.4.2) pro Prozessor eine Ausgabedatei für die Messungen von HPM erstellt. Es wird also die *Flip*-Rate für jeden einzelnen Prozessor angegeben. Um später mit der Threads-basierten Version vergleichen zu können, werden die Mittelwerte der *Flip*-Raten über alle Prozessoren ermittelt. Somit erhält man die *MFlip* pro Sekunde und pro Prozessor.

Flip-Rate in Mflip/sec pro Prozessor		
# Prozessoren	Master	1. Slave
1	3914.714	-
2	3776.361	3286.077
4	3206.281	2950.339
8	2977.666	2748.698
16	2866.913	2154.484
32	2096.007	2196.591

Tabelle 5.9: Flip-Raten pro Prozessor für *Cholesky_1* auf Master und erstem Slave, n = 357.911

In Tabelle 5.9 sind die *Flip*-Raten des Masters und des jeweils ersten Slaves aufgeführt. Dabei ist zu erkennen, dass der Master mehr Floating-Point-Operationen in einer Sekunde durchführt als der Slave (außer bei $p = 32$). Dies liegt daran, dass der Master, im Gegensatz zu dem Slave, zusätzlich für Sammeln und Zusammenfügen der Teilergebnisse zuständig ist. Dieser muss also mehr Aufwand betreiben, was bei gleicher Ausführungszeit zur Folge hat, dass auch dessen *Flip*-Rate höher ist, als die der Slaves.

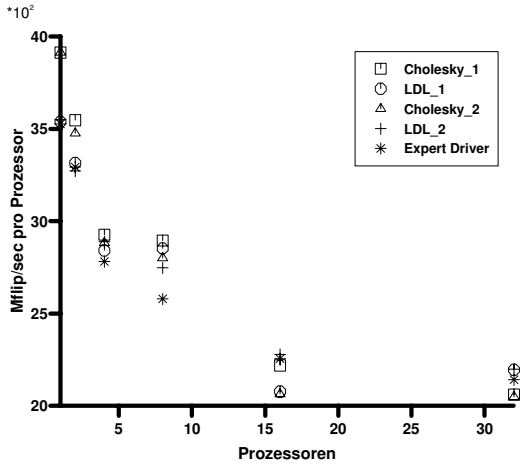

Abbildung 5.9: *Flip*-Raten pro Prozessor für die verschiedenen Verfahren, n = 357.911

Im Wesentlichen ist die *Flip*-Rate der verschiedenen Verfahren vergleichbar (Abbildung 5.9). Mit zunehmender Anzahl an Prozessoren fällt die *Flip*-Rate bei allen Verfahren deutlich. Dieses Verhalten ist vorhersehbar, da durch das Verteilen der Arbeit auf mehrere Prozessoren zum einen mehr Kommunikation entsteht und zum anderen sich Speicherzugriffe gegenseitig, wegen des gemeinsamen Speichers, behindern können. Auffallend ist, dass die Werte der Mflip/sec bei Cholesky schneller abnehmen, als bei den anderen Algorithmen. Die *Flip*-Rate ist bei Cholesky bei bis zu acht Prozessoren höher und ab acht Prozessoren niedriger als die der anderen Verfahren.

5.4.5 Speedup

Aufgrund der bisherigen Ergebnisse kann beim Speedup mit akzeptablen Ergebnissen gerechnet werden, was auch in Tabelle 5.10 zu erkennen ist.

# Prozessoren	*Cholesky_1*	LDL^T_1	*Cholesky_2*	LDL^T_2	*Expert Driver*
1	1	1	1	1	1
2	2.152	2.226	2.287	2.376	2.406
4	3.609	3.892	3.858	4.278	4.144
8	7.141	7.825	7.495	8.157	7.704
16	10.904	11.383	11.003	13.501	13.454
32	20.397	24.275	22.021	26.295	25.7667

Tabelle 5.10: Speedup der unterschiedlichen Verfahren, n = 357.911

Bei 1, 2, 4 und 8 Prozessoren werden diese jeweils fast vollständig ausgenutzt. Der Speedup liegt nahezu beim Maximum (Anzahl der momentan benutzten Prozessoren) und teilweise sogar darüber. Die Gründe sind in Abschnitt 5.3.5 aufgeführt.
Bei 16 und 32 Prozessoren wird deutlich, dass mehr Kommunikation stattfinden musste, da sich der Speedup nicht mehr dem Maximum annähert. Trotz allem sind die Werte noch in einem annehmbaren Toleranzbereich. Vor allem bei dem LDL^T_2-Verfahren und dem Expert Driver ist der Verlust durch die Kommunikation akzeptabel.

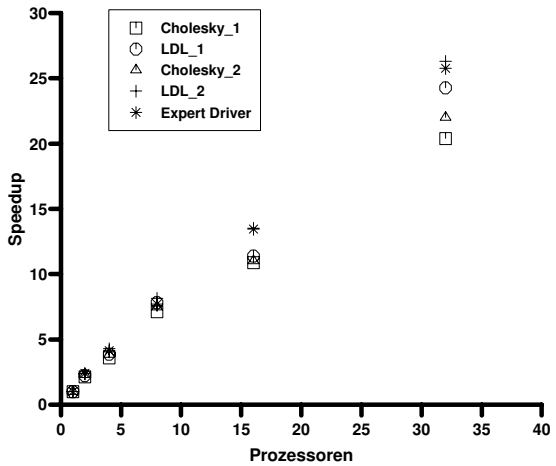

Abbildung 5.10: Speedup der unterschiedlichen Verfahren, n = 357.911

Der Speedup der verschiedenen Verfahren ist, mit Ausnahme von 32 Prozessoren, relativ gleich (Abbildung 5.10). *Cholesky_1* und *Cholesky_2* sind, wie in Abbildung 5.10 zu sehen, für kleine Prozessoranzahlen besser und für große Prozessoranzahlen schlechter als LDL^T_1 und LDL^T_2, wobei dafür auch das schnellere Abfallen der *Flip*-Rate bei den beiden *Cholesky*-Verfahren spricht. Daher skalieren die beiden LDL^T-Verfahren besser als die *Cholesky*-Verfahren.

5.4.6 Berechnung des seriellen Anteils

Mit Hilfe des Amdahlschen Gesetzes (Kapitel 5.2) soll nun der serielle Anteil bei den MPI-parallelen Berechnungen bestimmt werden. Da in den bisherigen Auswertungen festgestellt wurde, dass es keine großen Unterschiede zwischen den einzelnen Verfahren gibt, wird hier nur die Bestimmung des seriellen Anteils für das LDL^T_1-Verfahren durchgeführt.
Für den Geschwindigkeitsfaktor erhält man nach Amdahl:

$$S(p) = \frac{1}{f + \frac{1-f}{p}}$$

Aufgrund der Komplexheit dieses Problems habe ich nicht die analytische Lösung mit der Methode der kleinsten Fehlerquadrate durchgeführt, sondern das globale Minimum mit der GlobalOptimization-Toolbox [18] von Maple bestimmt. Die Toolbox basiert auf dem Simplex-Verfahren. Mit Hilfe der Messergebnisse für den Speedup aus Kapitel 5.4.5 und dieser Toolbox kann nun der serielle Anteil f bestimmt werden.
Als Ergebnis erhält man für das LDL^T_1-Verfahren für $n = 357.911$ den seriellen Anteil

$$f = 0.01151584.$$

Das heißt, der serielle Anteil beträgt etwa $1,15\%$ der Gesamtrechenzeit. Dies ist ein sehr niedriger Wert.

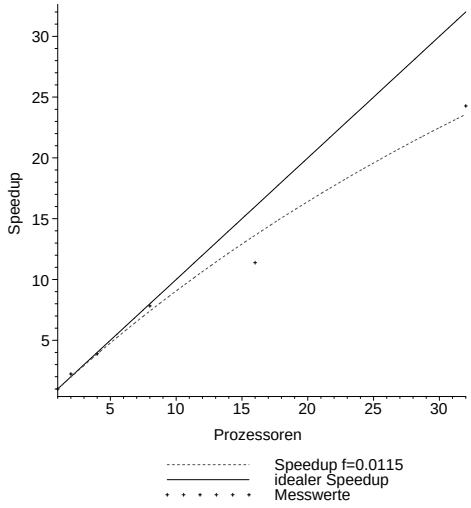

Abbildung 5.11: Speedup-Kurven für LDL^T_1, n = 357.911

Abbildung 5.11 zeigt, dass die Speedup-Kurve mit dem berechneten Amdahl-Wert für den seriellen Anteil im Vergleich zur idealen Speedup-Kurve ganz gut verläuft.

5.5 Vergleich zwischen Shared Memory Threads und MPI

Der Vergleich zwischen Threads-basierter und MPI-basierter Version wird nach denselben Kriterien wie in Kapitel 5.3 und 5.4 durchgeführt.

5.5.1 Anzahl der rechten Seiten

Die Laufzeit ändert sich bei unterschiedlicher Anzahl rechter Seiten nicht wesentlich. Die MPI-Version ist jedoch stabiler als die Threads-basierte, da bei Threads, wie bereits erwähnt, Speicherplatzprobleme innerhalb der Bibliothek WSMP auftreten. Bei der MPI-Version treten diese Probleme nicht erkennbar auf.

5.5.2 Problemgröße

Bei beiden Arten der Parallelisierung, MPI und Threads, zeigt sich, dass die fünf Verfahren (*Cholesky_1*, LDL^T_1, *Cholesky_2*, LDL^T_2 und *Expert Driver*) beim Vergleich der Laufzeit keine wesentliche Unterschiede aufweisen. Allerdings trat bei Verwendung von Threads ein Fehler in der Bibliothek auf, wie bereits in Kapitel 5.3.2 aufgeführt.

Laufzeiten *Expert Driver* in sec		
Dimension	Threads	MPI
132651	12.514	10.145
226981	32.248	28.329
357911	86.985	78.721
531441	190.855	167.767
753571	416.526	358.515
1030301	744.980	715.196

Tabelle 5.11: Laufzeiten Expert Driver, Threads- und MPI-basierte Parallelisierung, p = 4

In Tabelle 5.11 sind die Laufzeiten der beiden Parallelisierungsarten für den *Expert Driver* auf vier Prozessoren gegenübergestellt. Die Zeiten für die MPI-Version liegen dabei deutlich unter denen der Threads-Version. Bei den anderen Verfahren ist ein ähnliches Verhalten zu erkennen.
Beide Parallelisierungsmethoden, sowohl Threads als auch MPI, haben ihre Nachteile, die auf Problemen in der Bibliothek beruhen. Bei Verwendung von Threads tritt der oben erwähnte und in Kapitel 5.3.2 beschriebene Fehler bei bestimmten Problemgrößen und Anzahl an rechten Seiten auf. Hingegen besteht bei Verwendung von MPI das Problem, dass bei bestimmten Dimensionen und großer Anzahl an Prozessoren (16 und 32), die Programme mit einer Fehlermeldung abbrechen. Bei mehr als acht Prozessoren ist die MPI-Version deutlich überlegen.

5.5.3 Anpassen der Laufzeiten

Die Abhängigkeit der Ausführungszeit von der Matrixdimension konnte für die MPI-parallele Version mit Hilfe der Funktion

$$f(n) = a \cdot n^b$$

besser beschrieben werden als für die Version, die Threads zur Parallelisierung verwendet.

In Abbildung 5.12 ist zu sehen, dass der Graph der MPI-Version unter dem der Threads-Version liegt, das heißt, die Laufzeit bei Parallelisierung mit MPI ist geringer als die bei der Verwendung von Threads.
Die Funktionen für die in Abbildung 5.12 abgebildeten Graphen sehen folgendermaßen aus:

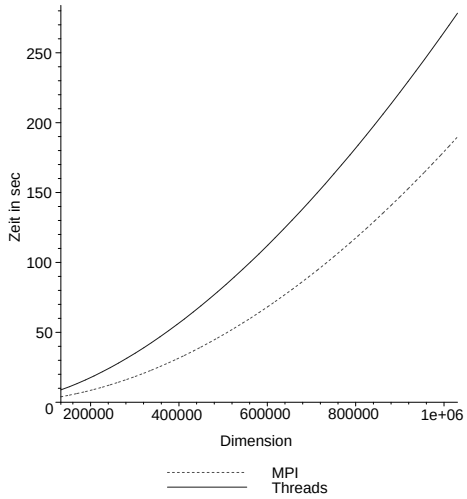

Abbildung 5.12: Laufzeiten von *Cholesky_2*, p = 16, Threads- und MPI-Version

$$f_{threads}(n) = 0.2114020513 \cdot 10^{-7} \cdot n^{1.682933710} \tag{5.1}$$

$$f_{mpi}(n) = 0.7619991283 \cdot 10^{-9} \cdot n^{1.895290185} \tag{5.2}$$

Bei beiden Gleichungen fällt auf, dass b bei Verwendung von Threads (5.1) kleiner ist als bei MPI (5.2). Damit ist auch die Ordnung niedriger, was auf ein besseres Verfahren hindeutet. Das heißt, auch wenn der Graph der MPI-Version in Abbildung 5.12 unter dem der Threads-Version liegt, liegt dieser ab der Dimension $n = 6.250.380$ unter dem von MPI. Das heißt, dass ab der Dimension $n = 6.250.380$ kann man erwarten, dass Threads weniger Zeit zur Ausführung brauchen als MPI. Bei den Messungen dieser Arbeit wird diese Dimension allerdings nicht erreicht.

5.5.4 Flip-Rate

Bei der MPI-Version wurde für jeden Prozessor die *Flip*-Rate bestimmt. Da diese voneinander abweichen können (siehe Kapitel 5.4.4), wurde der Mittelwert aller *Flip*-Raten berechnet. Um diese Mittelwerte nun mit den *Flip*-Raten bei Verwendung von Threads vergleichen zu können, wurden die Threads-*Flip*-Raten durch die jeweilige Anzahl der Prozessoren geteilt. Somit erhält man auch bei der Threads-Version eine *Flip*-Rate pro Prozessor.

Mflip/sec pro Prozessor		
# Prozessoren	Threads	MPI
1	3530.85	3533.89
2	3309.49	3380.02
4	2735.44	2970.07
8	2451.57	2951.17
16	1668.21	2366.26

Tabelle 5.12: *Flip*-Rate pro Prozessor, Threads- und MPI-Version für LDL^T_2, n = 531.441

In der Tabelle 5.12 sind die *Flip*-Raten der MPI-Version und der Threads-Version für das LDL^T_2-Verfahren dargestellt. Es wird nur dieses Verfahren betrachtet, da die *Flip*-Raten mit denen der übrigen Verfahren vergleichbar sind (Kapitel 5.3.4 und 5.4.4). Die *Flip*-Rate pro Prozessor liegt bei

Verwendung von MPI höher als bei Parallelisierung mit Hilfe von Threads. Bei einem und zwei Prozessoren ist der Unterschied noch sehr gering, aber ab vier Prozessoren nimmt die Differenz der *Flip*-Raten stark zu, so dass die *Flip*-Rate bei der Verwendung von Threads deutlich unter der bei der Verwendung von MPI liegt. Der Grund dafür ist, wie in Kapitel 5.3.4 beschrieben, dass es bei Parallelisierung mit Threads ab acht Prozessoren zu Speicherkonflikten kommt und daher die *Flip*-Rate kaum noch zunimmt. Bei Parallelisierung mit MPI hingegen tritt dieses Problem nicht auf.

5.5.5 Speedup

Der Speedup weist bei Verwendung von Threads größere Unterschiede zwischen den einzelnen Verfahren auf. Bei der MPI-Version war der Speedup bei allen ungefähr gleich. Zum Vergleich ist in Tabelle 5.13 der Speedup des LDL^T_1-Verfahrens aufgeführt, da dieses bei allen, bis auf 32 Prozessoren, den größten Speedup geliefert hat.

# Prozessoren	Threads	MPI
1	1.000	1.000
2	2.284	2.226
4	3.978	3.892
8	8.341	7.825
16	9.056	11.383
32	5.684	24.275

Tabelle 5.13: Speedup, Threads- und MPI-Version für LDL^T_1, n=357.911

Bis acht Prozessoren ist der Speedup der Threads-Version und der MPI-Version ungefähr gleich. Bei einer größeren Anzahl an Prozessoren nimmt der Speedup bei Verwendung von Threads kaum noch zu. Bei MPI hingegen steigt er weiter. Der Grund dafür sind ebenfalls Speicherkonflikte. Die Prozessoren können demzufolge bei MPI besser ausgenutzt werden als bei Threads, auch wenn bei MPI Verluste durch die Kommunikation entstehen.

Kapitel 6

Zusammenfassung

Ziel dieser Untersuchung war es, zu erfahren, auf welche Art man am effizientesten ein lineares Gleichungssystem mit dünnbesetzten symmetrisch positiv definiten Matrizen auf dem IBM Parallelrechner löst. Zur Lösung dieser Gleichungssysteme standen verschiedene Verfahren der Bibliothek WSMP zur Verfügung. Die Messungen wurden mit den Verfahren *Cholesky_1*, LDL^T_1, *Cholesky_2*, LDL^T_2 und dem *Expert Driver* durchgeführt. Es wurden dabei verschiedene Matrixdimensionen getestet und auf unterschiedlich vielen Prozessoren gerechnet. Außerdem wurde der Unterschied zwischen der Kommunikationsbibliothek MPI und der Shared Memory parallelen Programmierung mit Threads untersucht und aufgeführt. Es wurde zusätzlich geprüft, ob die Anzahl der rechten Seiten Einfluss auf die Rechenzeit der Programme hat.

Bei der Threads-basierten Parallelisierung zeigte sich kein wesentlicher Unterschied in der Rechenzeit der verschiedenen Verfahren und bei der Anzahl der rechten Seiten. Die aufgetretenen Abweichungen basieren auf einem Speicherzugriffsfehler in der Bibliothek WSMP. Allerdings war nur der Einsatz von bis zu acht Prozessoren sinnvoll. Erhöhte man die Prozessoranzahl, konnten keine wesentlichen Verbesserungen in der Laufzeit erreicht werden. Dies ist ein bekanntes Problem des IBM p690, auf dem die Rechnungen durchgeführt wurden.
Auch bei der Parallelisierung mit der Kommunikationsbibliothek MPI konnten keine größeren Unterschiede in den Rechenzeiten der verschiedenen Verfahren zur Zerlegung der Matrix festgestellt werden. Auch die Anzahl der rechten Seiten spielte keine Rolle in der Laufzeit. Dieses Verhalten war auch zu erwarten, da die Lösung des zerlegten Gleichungssystems mit viel weniger Rechenaufwand durchgeführt werden kann als die Zerlegung der Matrix.
Bis zu acht Prozessoren konnten mit MPI sehr gut und mehr als acht Prozessoren immerhin noch gut ausgenutzt werden. Der Anteil der Kommunikation hielt sich bei Verwendung von MPI in Grenzen. Allerdings trat auch bei der MPI-Version ein Fehler in der Bibliothek WSMP auf. Bei einigen Dimensionen und Anzahlen an rechten Seiten brach das Programm mit einer Fehlermeldung ab. Auch bei 32 Prozessoren, der maximalen Anzahl an Prozessoren, wurden bei einigen Dimensionen die Programme während der Ausführung abgebrochen.

Bei der MPI-Variante wurde zusätzlich zwischen zwei Varianten zur Erstellung der Matrix auf Basis der Hexe27-Elemente unterschieden. Zum einen wurde das Erstellen der Matrix auf dem Masterknoten und dem anschließenden Verteilen auf die weiteren Prozessoren und zum anderen das Erstellen der Teilmatrizen direkt auf den verschiedenen Prozessoren untersucht. Bei der Laufzeit der fünf Zerlegungs-Algorithmen konnte kein wesentlicher Unterschied zwischen beiden Methoden festgestellt werden.

Zuletzt wurde die Threads-basierte Parallelisierung mit der Parallelisierung unter Verwendung der Kommunikationsbibliothek MPI verglichen. Bei den betrachteten Matrixdimensionen und bis zu

acht Prozessoren waren beide Parallelisierungsarten vergleichbar. Beim Vergleich des Speedup, der Flip-Rate und dem Fit der gemessenen Ausführungszeiten wird klar, dass bei Verwendung von mehr als acht Prozessoren im untersuchten Dimensionsbereich mit MPI deutlich bessere Ergebnisse erzielt werden können als bei der Verwendung von Threads.
Allerdings sind die Fehler zu beachten, die sowohl bei Verwendung von Threads als auch bei MPI auftreten. Diese auf der Bibliothek basierenden Fehler können zu Einschränkungen in den Berechnungen führen. Da inzwischen eine neue Version der Bibliothek zur Verfügung steht, wäre es interessant, zu testen, ob diese Probleme inzwischen alle gelöst sind. Zusätzlich sind in der Dokumentation zur neuen Version einige Umgebungsvariablen erwähnt, die man ändern soll, da mit den Default-Einstellungen die Performance, insbesondere der Threads-basierten parallelen Version, schlecht ist. Es wäre interessant zu sehen, ob mit Änderung dieser Umgebungsvariablen auch eine bessere Skalierung der Threads-Version zu erreichen ist.

Bei den verschiedenen Verfahren ist kaum ein Unterschied in den Rechenzeiten festzustellen. Der Expert Driver liegt zwar in der Ausführungszeit etwas über den anderen Verfahren, aber bei ihm werden alle Schritte in einer Routine vorgenommen. Bei den anderen Verfahren wird zum Beispiel das Lösen des Dreieckssystems in seperaten Routinen durchgeführt. Das heißt, zu den betrachteten Rechenzeiten der anderen Verfahren kommen zusätzliche Rechenzeiten für beispielsweise symbolisches Faktorisieren oder Lösen hinzu und damit sind die Rechenzeiten aller Verfahren nahezu identisch, vor allem wenn man die Messungenauigkeiten mit in Betracht zieht. Da der Expert Driver einfacher zu nutzen ist (nur eine Routine) und in der Ausführungszeit der verschiedenen Verfahren kaum Unterschiede auftreten, ist der Expert Driver den anderen Verfahren vorzuziehen.

Anhang A

Verwendete Datensätze

1. **Harwell-Boeing-Matrizen**
   Als Datensätze wurden unter anderem Matrizen der Harwell-Boeing Sammlung von [19] verwendet. Diese Matrizen sind in einem bestimmten Format abgespeichert und stammen von unterschiedlichen Finite-Element-Problemen. Die verwendeten Datensätze sind alle dünnbesetzt und symmetrisch positiv definit.
   (a) Matrizen, die aus dem Eigenwertproblem

   $$K\boldsymbol{x} = \lambda M\boldsymbol{x}$$

   entstanden sind, wobei M dem verwendeten Datensatz entspricht.
      i. Die Matrix **BCSSTK25** hat eine Dimension von 15439×15439 und enthält 252.241 Nichtnull-Elemente.
      ii. **BCSSTM25** hat dieselbe Dimension wie BCSSTK25, nur dass außer der Diagonale alle Elemente gleich Null sind und ist damit sehr dünnbesetzt.
   (b) Matrizen, die einem statischen Problem zugrunde liegen.
      i. **BCSSTK16** ist 4.884×4.884 groß und hat 290.378 Einträge, die ungleich Null sind und ist damit dichter besetzt als **BCSSTK25**.
      ii. Die Matrix **BCSSTK17** enthält 428.650 Nichtnull-Elemente und hat eine Größe von
      10.974×10.974.
      iii. **BCSSTK18** mit einer Dimension von 11.948×11.948 und 149.090 Nichtnull-Elementen und ist damit sehr dünnbesetzt.
   (c) Die folgenden Matrizen resultieren aus Finite-Element-Diskretisierungen einer zylindrischen Schalenstruktur. Dabei sind die Enden des Zylinders frei.
      i. Eine Dimension von 90.449×90.449 und 2.455.670 Nichtnull-Elemente hat die Matrix **S3DKQ4M2**.
      ii. **S3DKT3M2** hat die gleiche Dimension, aber 1.921.955 Nichtnull-Elemente.
2. **Hexe27-Datensatz**
   Von den oben beschriebenen Harwell-Boeing-Matrizen gibt es nur beschränkt große Datensätze. Um bessere Erkenntnisse aus den Performance-Untersuchungen gewinnen zu können, wurde ein Programm entwickelt, das dünnbesetzte, symmetrisch positiv definite Matrizen anhand der Strukturmechanik mit Hexe27-Elementen erstellt. Dabei wird ein Quader mit 27 Knoten versehen (Abbildung A.1).
   Da das Besetzen der Matrizen nach natürlichen Gesichtspunkten sehr aufwendig wäre und auch einen großen Rechenaufwand darstellen würde, wurden die dieser Arbeit zugrunde liegenden Matrizen nach einem einfachen Schema berechnet. Das primäre Anliegen, einen

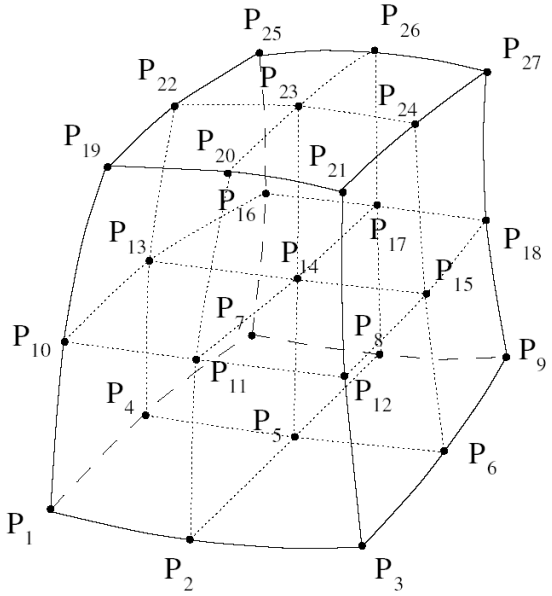

Abbildung A.1: Das Hexe27 Element

großen dünnbesetzten Datensatz zu erhalten, wurde erfüllt, wobei der Grad der Besetztheit bei ungefähr 0.01% liegt. Weitere Kriterien, wie die Wirklichkeitsnähe der Daten, sind für die Zeitmessungen dieser Arbeit von geringer Bedeutung und wurden daher nicht beachtet. Die Symmetrie ist bei Hexe27-Elementen ebenfalls gegeben. Damit die Matrix positiv definit ist, wurden die Elemente auf der Diagonalen um einen Faktor 100 größer als die anderen Elemente gewählt. Besetzt wurden die Elemente mit der Anzahl der Knoten, die das Element beeinflussen.
Als rechte Seiten wurden Vektoren der folgenden Form benutzt:
Falls nur eine rechte Seite verwendet wurde:

$$b_i = 1, \quad i = 1..n$$

Bei k rechten Seiten:

$$b_{i,j} = j - 1, \quad i = 1..n \quad j = 1..k$$

Anhang B

Tabellen zu den Messergebnissen

B.1 Messergebnisse der Threads-basierten Version

B.1.1 Laufzeiten Harwell-Boeing-Matrizen

Laufzeiten der Threads-parallelisierten Verfahren in sec					
Dimension	Cholesky_1	LDL^T_1	Cholesky_2	LDL^T_2	Expert Driver
4884	0.0326	0.0172	0.0539	0.0355	0.0505
bcsstk17	0.0530	0.0311	0.0516	0.0239	0.0412
bcsstk18	0.0393	0.0252	0.0413	0.0285	0.0463
bcsstk25	0.0798	0.0485	0.1150	0.0833	0.0118
bcsstm25	0.0076	0.0070	0.0076	0.0065	0.0355
s3dkq4m2	0.8781	0.6431	1.1103	0.8431	1.1741
s3dkt3m2	0.8577	0.6725	0.7482	0.5498	0.8156

Tabelle B.1: p =4, Harwell-Boeing-Matrizen

Laufzeiten der der Threads-parallelisierten Verfahren in sec					
Dimension	Cholesky_1	LDL^T_1	Cholesky_2	LDL^T_2	Expert Driver
bcsstk16	0.0342	0.0155	0.0440	0.0248	0.0383
bcsstttk17	0.0464	0.0182	0.0617	0.0380	0.0606
bcsstk18	0.0299	0.0168	0.0377	0.0250	0.0387
bcsstk25	0.0683	0.0302	0.0860	0.0586	0.0888
bcsstm25	0.0073	0.0067	0.0074	0.0065	0.0354
s3dkq4m2	0.5864	0.3573	0.6802	0.4206	0.6770
s3dkt3m2	0.5296	0.3545	0.6629	0.4294	0.7042

Tabelle B.2: p =16, Harwell-Boeing-Matrizen

Laufzeiten der der Threads-parallelisierten Verfahren in sec					
Dimension	Cholesky_1	LDL^T_1	Cholesky_2	LDL^T_2	Expert Driver
bcsstk16	0.0370	0.0151	0.0472	0.0216	0.0375
bcsstk17	0.0515	0.0184	0.0882	0.0543	0.0817
bcsstk18	0.0315	0.0164	0.0396	0.0240	0.0414
bcsstk25	0.0707	0.0323	0.1120	0.0781	0.1092
bcsstk25	0.0074	0.0069	0.0073	0.0065	0.0374
s3dkq4m2	0.6282	0.3561	1.0979	0.7599	1.1251
s3dkt3m2	0.5880	0.3671	0.6764	0.4357	0.7301

Tabelle B.3: p =32, Harwell-Boeing-Matrizen

B.1.2 Flip-Raten

Mflip/sec pro Prozessor					
# Prozessoren	Cholesky_1	LDL^T_1	Cholesky_2	LDL^T_2	Expert Driver
1	3675.116	3687.804	3693.040	3650.109	3451.73
2	3315.696	3387.541	3380.773	3336.802	3195.476
4	2687.700	2775.262	3185.313	2714.129	2546.768
8	2711.165	2914.917	2787.175	2953.514	2561.867
16	1414.985	1527.204	1375.857	1515.017	1301.424
32	736.823	793.044	656.216	753.881	658.749

Tabelle B.4: Flip-Raten, Matrix aus Hexe27-Elementen, n = 132.651

Mflip/sec pro Prozessor					
# Prozessoren	Cholesky_1	LDL^T_1	Cholesky_2	LDL^T_2	Expert Driver
1	3891.682	3557.928	3915.024	3522.013	3419.208
2	3682.874	3322.488	3564.218	3339.502	3247.554
4	3149.5855	2856.304	3182.399	3198.771	2979.493
8	3042.257	2983.791	3071.070	3026.309	2801.972
16	1655.071	1619.346	1617.542	1623.394	1528.155
32	812.948	831.949	777.360	799.413	747.139

Tabelle B.5: Flip-Raten, Matrix aus Hexe27-Elementen, n = 357.911

Mflip/sec pro Prozessor					
# Prozessoren	Cholesky_1	LDL^T_1	Cholesky_2	LDL^T_2	Expert Driver
2	3843.100	3396.564	3719.089	3316.209	3251.219
4	3091.181	2829.110	3064.881	2835.557	2766.398
8	3170.061	3061.090	3334.032	3140.802	3101.099
16	1717.389	1729.302	1737.083	1722.325	1659.962
32	835.756	846.995	836.288	847.032	822.233

Tabelle B.6: Flip-Raten, Matrix aus Hexe27-Elementen, n = 1.030.301

B.2 Messergebnisse der MPI-Version

B.2.1 Laufzeiten Harwell-Boeing-Matrizen

Laufzeiten der MPI-parallelisierten Verfahren in sec					
Dimension	Cholesky_1	LDL^T_1	Cholesky_2	LDL^T_2	Expert Driver
bcsstk16	0.0319	0.0173	0.0302	0.0176	0.0188
bcsstk17	0.0644	0.0399	0.0609	0.0388	0.0433
bcsstk18	0.0342	0.0198	0.0360	0.0220	0.0253
bcsstk25	0.0764	0.0491	0.0823	0.0538	0.0586
s3dkq4m2	0.0755	0.0565	0.0737	0.0561	0.0595
s3dkt3m2	0.8288	0.6419	0.8406	0.6667	0.6997

Tabelle B.7: p =4, Harwell-Boeing-Matrizen

Laufzeiten der MPI-parallelisierten Verfahren in sec					
Dimension	Cholesky_1	LDL^T_1	Cholesky_2	LDL^T_2	Expert Driver
bcsstk16	0.0232	0.0102	0.0214	0.0106	0.0118
bcsstk17	0.0355	0.0200	0.0309	0.0178	0.0200
bcsstk18	0.0212	0.0112	0.0210	0.0107	0.0131
bcsstk25	0.0382	0.0203	0.0382	0.0218	0.0243
s3dkq4m2	0.0354	0.0238	0.0331	0.0232	0.0251
s3dkt3m2	0.3214	0.2300	0.3111	0.2322	0.2548

Tabelle B.8: p =16, Harwell-Boeing-Matrizen

Laufzeiten der MPI-parallelisierten Verfahren in sec					
Dimension	Cholesky_1	LDL^T_1	Cholesky_2	LDL^T_2	Expert Driver
bcsstk16	0.0248	-	0.0222	0.0129	0.0133
bcsstk17	0.0357	0.0303	0.0333	0.0165	0.0191
bcsstk18	0.0238	-	0.0229	0.0130	0.0149
bcsstk25	0.0627	0.0296	0.0430	0.0317	0.0254
s3dkq4m2	0.0298	-	0.0273	0.0193	0.0204
s3dkt3m2	0.2992	0.1978	0.2861	0.1929	0.2097

Tabelle B.9: p =32, Harwell-Boeing-Matrizen

B.2.2 Flip-Raten

Mflip/sec pro Prozessor					
# Prozessoren	Cholesky_1	LDL^T_1	Cholesky_2	LDL^T_2	Expert Driver
1	3661.936	3674.231	3689.818	3661.531	3583.135
2	3188.170	3459.266	3057.143	3176.192	2901.342
4	2548.363	2742.377	2696.971	2894.428	2870.843
8	2402.463	2629.024	2148.954	3264.451	2280.967
16	1685.270	2813.237	1522.982	2816.790	2762.099
32	1635.233	1914.567	1638.566	1907.688	1848.129

Tabelle B.10: Flip-Raten, Matrix aus Hexe27-Elementen, n = 123.651

Mflip/sec pro Prozessor					
# Prozessoren	Cholesky_1	LDL^T_1	Cholesky_2	LDL^T_2	Expert Driver
1	3913.190	3538.057	3913.192	3548.236	3526.756
2	3547.403	3315.756	3475.721	3270.899	3292.3895
4	2925.775	2842.550	2882.491	2887.382	2780.922
8	2893.937	2851.608	2799.296	2746.947	2579.424
16	2218.075	2077.106	2062.624	2276.808	2255.389
32	2060.239	2196.434	2048.228	2197.030	2140.197

Tabelle B.11: Flip-Raten, Matrix aus Hexe27-Elemente, n = 357.911

Mflip/sec pro Prozessor					
# Prozessoren	Cholesky_1	LDL^T_1	Cholesky_2	LDL^T_2	Expert Driver
2	3841.296	3387.337	3827.203	3405.795	3420.421
4	2950.293	2714.373	2985.826	2710.094	2707.914
8	2952.160	2763.670	2997.196	2802.723	2784.617

Tabelle B.12: Flip-Raten, Matrix aus Hexe27-Elemente, n = 1.030.301

Literaturverzeichnis

[1] Götz Alefeld, Ingrid Lenhardt, Holger Obermaier,
Parallele numerische Verfahren
Springer Verlag (2000)

[2] *MPI homepage*
`http://www-unix.mcs.anl.gov/mpi/index.html`

[3] *OpenMP Fortran Application Program Interface*
`http://www.openmp.org/specs/`

[4] Dieter an Mey (2002)
Parallele Programmierung für Shared Memory-Rechner mit OpenMP
`http://www.rz.rwth-aachen.de/computing/events/material.php`

[5] *FZJ System Configuration*
`http://jumpdoc.fz-juelich.de/ibmsc/configuration/FZJ.html`

[6] Anshul Gupta (2000)
IBM Research Report
WSMP: Watson Sparse Matrix Package
Part 1 - direct solution of symmetric sparse systems
`http://www.alphaworks.ibm.com/tech/wsmp`

[7] *LoadLeveler*
`http://www.jumpdoc.fz-juelich.de/ibmsc/os`

[8] Luiz DeRose (2003)
Hardware Performance Monitor (HPM) Toolkit
Advanced Computing Technology Center
IBM Research
`http://jumpdoc.fz-juelich.de/ibmsc/software/HPM_2_4_4.html`

[9] Marlene Busch
Einführung in das Präsentationsgraphiksystem Gsharp
`http://www.fz-juelich.de/zam/docs/tki/t0306/t0306.html`

[10] Johannes Grotendorst
ComputerMathematik mit Maple, zweite Auflage
Schriftenreihe des Forschungszentrums Jülich (2004) (ISBN 3-89336-354-8)

[11] I.S. Duff, J.K. Reid
The multifrontal solution of indefinite sparse symmetric linear equations
ACM Trans. Math. Software, 9 (1983), pp.302-325

[12] P.R. Amestoy, I.S. Duff, J.-Y. L'Excellent (1998)
Multifrontal Parallel Distributed Symmetric and Unsymmetric Solvers
`http://epubs.cclrc.ac.uk/work-details?w=29735`

[13] Patrick R. Amestoy, Chiara Puglisi (2002)
An unsymmetrized multifrontal LU factorization
`http://epubs.siam.org/sam-bin/getfile/SIMAX/articles/37537.pdf`

[14] Joseph W. H. Liu (1992)
The multifrontal method for sparse matrix solution: theory and practice
SIAM Review Volumn 34, Number 1

[15] Jürgen Schulze (2000)
Faktorisierung dünnbesetzter, positiv definiter Matrizen
`http://webdoc.sub.gwdg.de/ebook/e/2002/schulze/disschul.pdf`

[16] P.R. Amestoy, I.S. Duff, T.A. Davis (2004)
AMD, an approximate minimum degree ordering algorithm
`http://www.cise.ufl.edu/research/sparse/amd/v1.1/amdalgo.pdf`

[17] Bodo Kalthoff (2002)
Einsatz von algorithmischen Skeletten im Scheduling massiv paralleler Systeme
`http://ubdata.uni-paderborn.de/ediss/17/2002/kalthoff/disserta.pdf`

[18] *Global Optimization Toolbox for Maple*
`http://www.maplesoft.com/toolboxes/globaloptimization/index.aspx`

[19] *Matrix Market*
`http://math.nist.gov/MatrixMarket`